McGill University
School of Computer Science

Taskell: a concurrent constraint programming language

Clément Pellerin

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Masters of Science

Abstract

Taskell is an instance of the concurrent constraint programming framework cc. The framework is parameterized by a choice of constraint system. The constraint system of Taskell is the set of finite trees with equality. The choice of constraint system makes Taskell similar to concurrent logic programming languages. When computing with partial information the notion of reading and writing memory becomes incoherent. The framework replaces these operations by ask and tell respectively. We hope to understand this new paradigm by studying implementations of cc languages. Taskell is a parallel implementation of a cc language written in Concurrent ML.

Résumé

Taskell est un langage de programmation concourant avec contrainte membre de la famille cc. Le système de contraintes de Taskell comprend les égalités d'arbre fini. Taskell est similaire à un langage concourant de programmation logique à cause de son système de contraintes. Les notions de lecture et d'écriture de la mémoire ne sont pas cohérentes lorsque l'on calcule avec de l'information partielle. La famille cc remplace ces opérations par *ask* et *tell* respectivement. Nous esperons comprendre mieux ce nouveau paradigme en étudiant l'implantation de langage cc. Taskell est une implantation parallèle d'un langage cc écrite en ML Concourant.

To Lise, Pierre, and Chris
for making this possible.

Acknowledgments

This research was funded by NSERC, FCAR and the School of Computer Science.

I wish to thank Prof. Chris Paige for believing in me when it counted. I could not have finished my masters at this time if he had not been there. He went beyond his duty as a professor on many occasions. He has grown to be a friend more than an advisor and I wish him well.

I'd like to thank Prof. Prakash Panangaden for his joyful assistance and guidance. When I was unhappy with my topic, he offered this one. I offer him this implementation.

Thanks to Prof. Laurie Hendren for teaching me ML and showing me her compiler written in ML. I also thank her for the role model she gave me. She's the proof someone like me can one day be a professor.

I would like to thank my fellow student Gilles Pesant now at Université de Montréal. He read my manuscript and made numerous corrections.

Thanks to Prof. Gerald Ratzer for letting me be his teaching assistant for so long. I also wish to thank all the other individuals that were part of my life at McGill.

Special thanks go to John Reppy who served as my **CML** guru. It is not always that you have the opportunity to talk to the original designer of the language you are using. His timely response to my electronic messages were always beneficial. When all else failed, I could always count on him to solve my problem. I remember the time when I thought there was a bug in **CML**, John proved me wrong and he even told me how to fix mine!

Je remercie ma tendre amie Nicole Lachance qui m'a rendu heureux pendant tout le temps nécessaire à ces travaux et pour longtemps encore j'espère. J'ai écrit ces quelques lignes en français pour être certain qu'elle puisse les lire. Je t'ai promis si souvent que j'étais sur le point de finir. C'est fait.

Contents

List of Figures

Chapter 1

Introduction

1.1 Subject of the Thesis

This thesis describes a prototype implementation of a concurrent constraint programming language. The implementation is written in the parallel programming language **CML**. The language we implemented is a member of the cc (for concurrent constraint) framework investigated by Saraswat[Sar89]. We say cc is a framework because it forms a family of languages. The framework is parameterized by a choice of constraint system. Each instance has its specific constraint system but it shares the same concurrent process language. This thesis describes the implementation of an instance of cc where the constraint system is unification over finite trees.

A program written in a cc language computes with possibly incomplete information. As execution progresses, consistent information is added, never retracted. This gives rise to a new paradigm that subsumes nondeterminate data-flow and concurrent logic programming among others. We hope to understand the feasibility of concurrent programming in this style. We are trying to understand this new paradigm by implementing it and running programs in our implementation. The experience gained in writing cc programs will help improve any future implementation efforts.

Given that cc languages are concurrent, it is particularly instructive to study the interaction between processes. Concurrent programs are difficult to design and implement. There are more opportunities to make mistakes. Once a mistake is present it is harder to find and correct. Our implementation can produce a trace of an execution run, from which the concurrency is readily apparent. The programmer can grasp the behavior of a program from studying its trace.

Our cc language is called Taskell (an anagram of the words ask and tell which form the basic communicative primitives in the cc paradigm). We chose the syntax to be faithful to the basic ask-tell notation. The syntax should be clear to anyone who has read the basic papers on the cc paradigm.

The implementation is written in Concurrent ML or **CML** for short[Rep90] This is also the target language of our compiler. The compiler uses a recursive descent parser with some error recovery. When the parse tree is constructed without error, it is translated into **CML** in one pass. The output can be executed once it is linked with the run-time system. The run-time system is responsible for process management and constraint solving. **CML** does not have an implementation on a parallel machine, though one is in preparation. When it does, we will automatically have a parallel implementation of our language working on a parallel machine.

We have tested our system with a set of sample programs. Some of them are given in chapter 5. In an appendix we give a complete example, with the source, the output of the compiler, and the trace of the execution. One can get an overall idea of how the system works from this example.

The origins of constraint programming date back to the sixties. In a sense, Sutherland's Sketchpad graphical program[Sut63] was based on constraints. Later, the language in Steele's thesis[Ste80] and work by Borning on THINGLAB[Bor79, Bor81] elaborated the ideas of constraint programming. These early efforts were successful in so far as they went, but lacked generality to be truly general-purpose languages. The work of Leler is special in this regard, as the goal of his language Bertrand is to simplify the creation of new constraint languages using an augmented rewriting system[Lel88]. The traditional branch of constraint programming is still an active area of research.

The late 1980's has seen the introduction of constraints to the field of logic programming. The constraint logic programming scheme CLP arose from a need to provide a theoretical foundation to the many extensions of Prolog[JLM86]. The concurrent constraint programming language framework cc can be viewed as a gener-alization of CLP to the parallel case, but in fact it arose as an attempt to give a clean semantics to the various parallel extensions of Prolog: Concurrent Prolog[Sha87], PARLOG[CG86, Gre87, Col89] and Guarded Horn Clauses[Ued86a, Ued86b].

1.2 Chapter Summaries

In chapter 2 we review logic programming. Many extensions of logic programming have been proposed, most have only an operational semantics. This is unfortunate because the declarative semantics of logic programming is one of its great alleged advantages. It is possible to give a declarative semantics to many of these extensions by introducing the notion of constraints in logic programming. The constraint logic programming scheme offers a declarative semantics to many extensions that were once only explained operationally.

Chapter 3 considers the concurrent logic programming languages. These languages view the execution of a logic program as a collection of processes that communicate with each other through shared variables. Processes synchronize on the availability of data, i.e., they wait until a variable gets instantiated. These languages have committed-choice nondeterminism also known as don't-care nondeterminism. The commit operator is a type of cut generalized to the parallel case.

Chapter 4 explains the cc framework for concurrent constraint programming. In a von Neumann language, the memory is a valuation mapping variables to values. In this framework, the memory is replaced by a store, which is a constraint on the variables that may only partially specify them. With such a store, the notions of reading and writing variables become incoherent. For example, a variable in the store might be unconstrained, reading this variable should return infinitely many values. Assigning a value to a variable might have an impact on many other variables through the constraints that were imposed between them and the variable we assigned. Instead cc replaces read and write by *ask* and *tell*. A process can ask if a constraint c is already entailed by the store. It will succeed if c is entailed, suspend if c may or may not be entailed, and it will fail precisely when the store already entails $\neg c$. A process can tell a constraint c. The store is augmented with this new constraint, it becomes the conjunction of the old store with c. The store must remain consistent. The program is considered erroneous if a tell makes the store inconsistent.

Chapter 5 describes the syntax of Taskell. We relate our syntax with the one used in Saraswat's thesis[Sar89]. Our syntax tends to spell out the names of the operators in full. Procedures are relations as in logic programming. We show a set of sample programs to get a feel for programming in Taskell.

Chapter 6 describes our implementation strategy. The Taskell compiler and its output are written in Concurrent ML[Rep91]. **CML** is a very-high level language. This helped the implementation because we could rely on **CML** for process management, inter-process communication and garbage-collection. Our compiler uses recursive descent to parse a Taskell program in one pass. The abstract syntax tree produced is translated into **CML** in a second pass. The output can be linked with the run-time system to form an executable. Parallelism in Taskell is very fine-grain. The processes in **CML** are a perfect match because they are very light-weight[Rep91].

In the last chapter we give our conclusions. Following the last chapter is an appendix showing a session with the system. A sample program is taken from chapter 5. The translation of the compiler is shown, and the trace of its execution is given. The reader is referred to chapter 6 for an explanation of the appendix.

Chapter 2

Constraint Logic Programming

In this chapter we look at constraint logic programming or CLP. We start by reviewing the fundamentals of logic programming. We then survey some extensions that were proposed in order to alleviate some shortcomings present in logic programming. Two extensions specifically attract our attention: incorporating functional programming into logic programming and unification of infinite rational terms. These extensions are very ad hoc, most were presented by giving their operational semantics with no correspondence in logic. The accomplishment of CLP is to give a theoretical foundation to these diverse ideas. CLP is a scheme, it is parameterized by a choice of constraint system. Many logic programming languages can be reconstructed as instances of CLP with a suitable constraint system. For example, Colmerauer only gave an operational semantics to Prolog II, but Prolog II is a CLP language. Its constraint system is equality and disequality over rational trees. Hence, the denotational semantics of Prolog II is given by that of CLP.

2.1 Standard Logic Programming

Logic programming studies the use of first-order logic for computation. It started as an application of automated theorem proving technology to the design of a programming language. A major advance was made in 1965 when Robinson proposed resolution as the single inference rule in a first-order theory[Rob65]. Resolution is well-suited for computer applications because it needs only one inference rule and it works by refutation which means it can be implemented with backwards chaining. Though Robinson is given credit for resolution, most of the ideas were anticipated in the proof-theoretic investigations of Herbrand[Her30].

Resolution needs sentences to be written as a set of clauses. In 1971, Kowalski observed that we could restrict logic programs to a set of Horn clauses without a great loss in expressive power. This improved the efficiency remarkably. In the early seventies Colmerauer and his team, influenced by Kowalski, developed Prolog (for PROgramming in LOGic). Prolog is the first and the most widely known logic programming language.

This section is based on the survey of Shapiro[Sha89]. We describe the syntax of a logic program. A term is a variable (e.g. X, Y) or a function symbol of arity $n \geq 0$ applied to n terms (e.g. c, $f(X, 1, 2)$). An atom is a formula of the form $p(T_1, \ldots, T_n)$ where p is a predicate of arity n and $T_1, \ldots, T_n$ are terms. A definite clause (or clause for short) is a formula of the form $A \leftarrow B_1, \ldots, B_n \quad (n \geq 0)$ where A is an atom and $B_1, \ldots, B_n$ is a sequence of atoms. A is called the clause head and $B_1, \ldots, B_n$ its body. A unit clause is a clause with $n = 0$, we represent it with $A \leftarrow \mathit{true}$. A logic program P is a finite set $C_1, \ldots, C_n$ of definite clauses. A goal is a sequence of atoms $B_1, \ldots, B_n$. Each atom in a goal is called a goal atom. The procedure for a predicate p of arity n is the set of clauses in the program P whose head is an atom formed with the predicate p of arity n. To make a loose comparison, we can say function symbols are the data structures of logic languages, atoms are the statements, clauses are the procedures and goals are statements we ask the system to prove.

Consider a clause $A \leftarrow B_1, \ldots, B_n \quad (n \geq 0)$. Let $x_1, \ldots, x_k$ be the variables appearing in the head and let $y_1, \ldots, y_l$ be the variables appearing in the body but not in the head. The clause can be read: for all $x_1, \ldots, x_k$, A if there exists $y_1, \ldots, y_l$ such that B_1 and ... and B_n. Consider a goal $B_1, \ldots, B_n$. Let $y_1, \ldots, y_l$ be the variables appearing in the goal. It can be read: there exists $y_1, \ldots, y_l$ such that B_1 and ... and B_n. Consider the program $C_1, \ldots, C_n$, where each C_i is a definite clause. It can be read as the conjunction of the clauses: C_1 and ... and C_n.

Kowalski was the first to give a procedural interpretation to logic programs. The theorem prover is viewed as an interpreter for procedure calls initiated by a goal statement. Each procedure, when called, is free to call on other procedures. A computation stops when no more procedure calls can be made. A definite clause $A \leftarrow B_1, \ldots, B_m$ can be viewed as a procedure definition. The variables appearing in the head are the formal parameters. The variables appearing in the body but not in the head are the local variables. In a procedural language, this clause is equivalent to:

```
**procedure** $A(x_1, \ldots, x_k)$
**var** $y_1, \ldots, y_l$
**begin**
    **call** $B_1$
    ⋮
    **call** $B_n$
**end**
```

Similarly, the attempt to prove a goal $B_1, \ldots, B_n$ can be viewed as a sequence of procedure calls. Unification is used for parameter passing, assignment, data selection and data construction.

We will describe the semantics of a logic program with a transition system. In order to express a single transition step we need the definitions of substitution, unification and most general unifier (mgu). A substitution is a function from variables to terms that is the identity except on a finite subset of variables. A substitution θ can be written as $\{X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\}$, where the X_i's are variables and the T_i's are terms, $X_i \neq T_i$, and $X_i \neq X_j$ for $i \neq j$. $\{X_i \mapsto T_i\}$ is called a binding for X_i and X_i is said to be bound to T_i in the substitution. The application of θ to the variable X is denoted $X\theta$. We have $X\theta = X$ if $X \notin \{X_1, \ldots, X_n\}$, otherwise X is X_i and $X\theta = T_i$. A substitution is generalized to a function from terms to terms. Let T be a term, if T is the variable X_i then $T\theta = T_i$, if T is a variable not belonging to $\{X_1, \ldots, X_n\}$ then $T\theta = T$, otherwise T is some function symbol f of arity k applied to its arguments $A_1, \ldots, A_k$, and $T\theta = f(A_1\theta, \ldots, A_k\theta)$. Applying θ to T is called instantiating T by θ, $T\theta$ is called an instance of T.

A substitution θ is a unifier of two terms T_1 and T_2 if $T_1\theta = T_2\theta$. The unification algorithm takes two terms and finds a unifier for these two terms if one exists. The composition of two substitutions θ_1 and θ_2, written $\theta_1\theta_2$ is the function resulting from applying θ_1 first and then θ_2 to the result. Let $\theta_1 = \{X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\}$, and $\theta_2 = \{Y_1 \mapsto S_1, \ldots, Y_k \mapsto S_k\}$, then $\theta_1\theta_2 = \{X_1 \mapsto T_1\theta_2, \ldots, X_n \mapsto T_n\theta_2, Y_1 \mapsto S_1, \ldots, Y_k \mapsto S_k\}$ where in this set bindings of the form $X_i \mapsto X_i$ are deleted and bindings for Y_i are omitted if $Y_i = X_j$ for some j. These restrictions make sure $\theta_1\theta_2$ fulfills the conditions in the definition of substitution. Note that composition of substitutions is associative so parentheses are not needed in expressions like $\theta_1 \cdots \theta_n$.

A unifier μ is a most general unifier of two terms T_1 and T_2 if for every unifier θ of T_1 and T_2, we can find a substitution σ such that $\theta = \mu\sigma$ The unification algorithm finds most general unifiers. The function mgu(T_1,T_2) returns a most general unifier of T_1 and T_2 if one exists and fails otherwise.

The occurs-check is a step in the unification algorithm. It is required to guarantee termination. The occurs-check determines if a variable is present in a term. Before the unification algorithm binds a variable X to a term T, it uses the occurs-check to see if X occurs in T. If it does, the unification fails, otherwise X is bound to T.

Computation in logic programming is a search for a proof. The program P is the set of axioms, the initial goal G is what needs to be proven. The proof can be extracted from the path the computation went through.

A state of the computation is a pair $\langle G; \theta\rangle$ where G is a goal and θ is a substitution. The initial state is $\langle G; \varepsilon\rangle$ where G is the initial goal and ε is the empty substitution or identity function. There are two kinds of terminal states depending on the result of the computation. The computation is successful if it terminates in a state $\langle \text{true}; \theta\rangle$. The substitution in the terminal state restricted to the variables of the initial goal is the answer substitution. The initial goal instantiated by the answer substitution is a logical consequence of the program as can be concluded from the proof. The computation fails if it ends in a state $\langle \text{fail}; \theta\rangle$. In that case, the computation could not be extended to form a proof of the initial goal either because the initial goal is unprovable or possibly because a bad choice was made earlier which does not lead to a proof.

We denote by $\langle G;\theta\rangle \rightarrow \langle\hat{G};\hat{\theta}\rangle$ the transition from state $\langle G;\theta\rangle$ to state $\langle\hat{G};\hat{\theta}\rangle$. In resolution, one needs to rename the variables in a clause so it doesn't have any variables in common with the goal. This is called renaming the variables apart, and the resulting clause is called a variant of the clause. This ensures we can reuse the same clause more than once in the same computation.

There are two transition rules:

Reduce

$\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \rightarrow \langle(A_1,\ldots,A_{i-1},B_1,\ldots,B_k,A_{i+1}\ldots,A_n)\hat{\theta};\theta\hat{\theta}\rangle$

if $\text{mgu}(A_i,A)=\hat{\theta}$ for a renamed-apart clause $A \leftarrow B_1,\ldots,B_k$ of P.

Fail

$\langle A_1,\ldots,A_n;\theta\rangle \rightarrow \langle\text{fail};\theta\rangle$

if for some i, and for every renamed-apart clause $A \leftarrow B_1,\ldots,B_k$ of P, we have $\text{mgu}(A_i,A)=\text{fail}$.

A computation of a program P on a goal G is a finite (or infinite) sequence of states $s_1,s_2,\ldots$ such that $s_1=\langle G,\varepsilon\rangle$ is the initial state, $s_i \rightarrow s_{i+1}$ for every consecutive pair of states and if the sequence is finite of length k then s_k is a terminal state $\langle\text{true},\theta\rangle$ or $\langle\text{fail},\theta\rangle$.

The transition system is non-deterministic in the sense used in automata theory, i.e., one is interested in the existence of one successful computation path. Failure means that all paths fail. This is implemented by depth-first search of the state space with backtracking on failure.

2.2 Extensions of Logic Programming

Researchers have proposed many extensions to logic programming, see for instance [DL86]. We will concentrate on three extensions: unification with equality, introducing functions in logic programming and unification over rational trees. Our discussion remains superficial because we are less interested in the extensions than showing the diversity of ideas that are encompassed by CLP.

Unification with equality. Equality in Prolog is very weak because it is based on unification. Two terms are equal if they are syntactically the same. Unfortunately this means two terms that are semantically equal but syntactically different will not unify. For example, $2 + 2 = 1 + 3$ will not unify even though 4 is equal to 4. The unification algorithm can often be generalized to unify two terms whenever they are equal in some equational theory. This algorithm is called E-unification.

E-unification is achieved by adding a term rewriting system to the Prolog engine. Computation proceeds by narrowing, a process similar to alternating resolution and term rewriting. The user can write his own rules to guide the rewriting process. Unfortunately, this can lead to problems because there is no most general unifier. For example, it is not possible to write rules for associativity. Some systems assume the term rewriting rules are confluent. EQLOG is an example of Prolog with equality[GM86].

Introducing functions into logic programming Functional programming is based on two key concepts: reduction of expressions to a normal form and λ-abstraction for abstracting over expressions to make functions. Functions are first-class values, they can be bound to a variable, passed around as arguments and returned as the result of a function.

Functional programming has some advantages over logic programming[BL86]: the functional formalism is more readable than the relational one, the reduction to normal form is backtracking-free, lazy evaluation can handle infinite streams, type systems are more advanced and λ-abstraction allows the creation of anonymous functions on-the-fly.

Logic programming has some advantages too: it can compute with incomplete data structures, i.e., terms that contain unbound variables, predicates are multidirectional so arguments are not strictly input or output and it can express constraints better because a constraint is nothing more than a relation.

Prolog has a few built-in functions, mostly for arithmetic expressions. The program has to trigger the evaluation of an expression with the 'is' predicate. The programmer has to make sure every variable in the expression is already bound, otherwise the program is aborted with an error. The programmer cannot add to the set of built-in functions.

Introducing functions into logic programming tries to alleviate these problems while retaining its advantages. Bellia and Levi survey previous efforts in that direction[BL86]. One possible solution is to use semantic unification which is essentially a form of narrowing. FUNLOG is a logic programming language with functions based on semantic unification[SY86].

Unification over rational trees. The occurs-check is required by the unification algorithm to forbid the creation of bindings of the form $X = f(X)$ where a variable is assigned a term containing an occurrence of itself. The solution of those equations is an infinite term. In our example, X is the term $f(f(f(\ldots)))$. Most Prolog implementations do not perform the occurs-check for efficiency reasons. In those implementations, it is possible to create infinite terms. Unfortunately, the unification algorithm may fall into an infinite loop when unifying two infinite terms. The same problem happens when it comes time to print the value of an infinite term. For these reasons, logic programmers are taught to avoid creating infinite terms.

The type of infinite terms appearing in logic programs is called a rational tree. A rational tree has the important property that it has only a finite number of distinct subtrees. If you merge all the nodes in a rational tree that are the root of the same subtree, you will be left with a finite graph. A rational tree is infinite if and only if there is a cycle in the graph. In unification, a cycle forms when a variable v is bound to a term containing v.

Colmerauer was looking for a way to justify the omission of the occurs-check He gave a unification algorithm for rational trees[Col82, Col84, Col86]. It does not need the occurs-check because it handles infinite rational trees correctly. The algorithm is guaranteed to terminate because it looks at every different subtree only once and there is only a finite number of those. Subsequently his algorithm has been improved. The latest algorithm[Jaf84, MR84] is a simplification of the linear unification algorithm of

Martelli and Montanari[MM82].

To explain his algorithm, Colmerauer recasts the problem of unifying two terms into the problem of solving a system of equations. For example, unifying $p(a,b)$ with $p(X,Y)$ given that $X = a$ and Y is unbound is the same thing as solving the system of equations: $\{X = a,\ p(a,b) = p(X,Y)\}$. This system is solvable because it reduces to $\{X = a,\ Y = b\}$. A nice theory of unification in this setting is given in [LMM88].

Colmerauer implemented his algorithm in a Prolog system. He called his language Prolog II. A goal in Prolog II is a sequence $A_1, \ldots, A_n$ where A_i is an atom or an equation $T_1 = T_2$. A state of the computation is a pair $\langle G; E\rangle$ where G is a Prolog II goal and E is a system of equations in solved form (for a definition of solved form see [Col84]). The initial state is $\langle G, \varepsilon\rangle$ where G is the initial goal and ε is the empty system of equations. Prolog II replaces the unification of an atom A_i with the head of a clause A by the equation $A_i = A$. The unification succeeds if $\hat{E} = E \cup \{A_i = A\}$ is solvable. $\hat{E}$ is kept in solved form. Equations in a goal are simply added to the system of equations provided the augmented system is solvable. There are three transition rules:

Reduce atom

$\langle A_1, \ldots, A_i, \ldots, A_n; E\rangle \rightarrow \langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1} \ldots, A_n; \hat{E}\rangle$

if A_i is an atom,

$A \leftarrow B_1, \ldots, B_k$ is a renamed-apart clause of P.

$\hat{E} = E \cup \{A_i = A\}$ is solvable,

and $\hat{E}$ is in solved form.

Reduce equation

$\langle A_1, \ldots, A_i, \ldots, A_n; E\rangle \rightarrow \langle A_1, \ldots, A_{i-1}, A_{i+1} \ldots, A_n; \hat{E}\rangle$

if A_i is an equation $T_1 = T_2$,

$\hat{E} = E \cup \{T_1 = T_2\}$ is solvable,

and $\hat{E}$ is in solved form.

Fail

$\langle A_1, \ldots, A_i, \ldots, A_n; E\rangle \rightarrow \langle \text{fail}; E\rangle$

if A_i is an atom,

and for every renamed-apart clause $A \leftarrow B_1, \ldots, B_k$ of P,

$E \cup \{A_i = A\}$ is not solvable,

or A_i is an equation $T_1 = T_2$,

and $E \cup \{T_1 = T_2\}$ is not solvable.

The computation is successful if it terminates in a state $\langle \text{true}; E \rangle$. The answer is the value E assigns to the variables of the original goal. The computation fails if it ends in a state $\langle \text{fail}; E \rangle$. This transition system like the one for standard logic programming is non-deterministic. The implementation of Prolog II shares its backtracking mechanism with Prolog.

Prolog II is even more expressive since it allows disequality as a primitive. A disequality is a formula of the form $X \neq Y$. In Prolog, if you want to say X should not be $p(a, b)$ you have to write $not(X = p(a, b))$. This introduces problems with negation as failure[Nai86]. Good Prolog implementations wait until X has a value before checking if it unifies with $p(a, b)$. In Prolog II, you can say dif(X,$p(a, b)$) to express that X and $p(a, b)$ are different. If later you say $X = p(Y, Z)$ then the equation solver knows either $Y \neq a$ or $Z \neq b$. Disequalities are kept as long as they are not guaranteed to hold. It is possible for a goal to terminate before enough information is known to make that guarantee. In that case, disequalities are part of the answer. For example, the answer for the goal ?-dif(X,$p(a, b)$), $X = p(Y, Z)$, is $X = p(Y, Z)$, $Y \neq a$ or $Z \neq b$.

A state of the computation is really a triple $\langle G; E; I \rangle$ where E and G are as before and I is a system of disequalities. The transition rule for the atom dif(T_1,T_2) adds $T_1 \neq T_2$ to I if this is consistent with E, and fails otherwise. The system I may be simplified every time there is an equation added to E. A successful computation terminates in a state $\langle \text{true}; E; I \rangle$. The answer is the value E assigns to the variables in the original goal subject to the restrictions in I. We do not give this transition system because the next section has a more general treatment.

2.3 Constraint Logic Programming

Most extensions to logic programming are very ad hoc and are explained operationally with little connection to logic. The constraint logic programming scheme CLP is an attempt to unify these extensions into a formal theory[JLM86]. CLP is parameterized by a domain of values D and its associated constraint system C. It can be viewed as a family of languages each one on a different domain. For example, the first CLP language implemented as such was CLP($\mathcal{R}$)[HJM$^+$87]. Its domain is the set of real numbers and the constraints that can be put on them are equalities and inequalities.

Computing with constraints has two great benefits over standard logic programming. First the values can be taken directly from the domain, they need not be coded syntactically. For example, in Prolog, a rational number x/y can be coded as $r(x, y)$. None of the built-in arithmetic functions will work on this representation, not even for numbers of the form $x/1$. Furthermore, $r(1,5)$ is not equal to $r(2,10)$ because they are syntactically different. The rational numbers follow the usual arithmetic laws in a CLP language with rational numbers in the domain. The second benefit is the pruning of the search space. Backtracking detects failures *a posteriori*, once it is already too late. Worse, it goes back to the latest choice point regardless of the cause of failure. The interpreter will often rediscover the same failure again and again until it finds the cause of the failure. A CLP language can use the constraints to cut down the search space *a priori*, before a failure happens. The interpreter may find that a set of constraints is unsolvable and backtrack immediately instead of searching beyond that point. This idea was used extensively in the language CHIP to solve constraint satisfaction problems[Hen89].

The parameterization of CLP is important because all members of the scheme share the same semantics. It turns out that unification is not central to Herbrand's theorem. Jaffar, Lassez and Maher were able to prove a Herbrand-like theorem replacing unification by constraint satisfaction[JL87]. The theorem makes very few assumptions on the constraint system. In particular, it does not depend on the kind of constraint system used. This is why the semantics can be given for a class of languages all at once.

The properties needed by the theorem are: solution compactness and satisfaction completeness[Coh90]. A constraint system is solution compact if every value in the domain can be represented by a possibly infinite set of constraints. In other words, the constraints must be fine enough so we can talk about every element in the domain. We also require the complement of a constraint to be representable as the union of a possibly infinite set of constraints. A constraint system is satisfaction complete if every constraint is either provably satisfiable or provably unsatisfiable. This is needed by the refutation procedure, if a set of constraints is unsatisfiable, it is guaranteed to be recognized as such in finite time.

A CLP goal is a sequence $A_1, \ldots, A_n$ where A_i is either an atom or a constraint. A state of the computation is a pair $\langle G; \sigma \rangle$ where G is a goal and σ is a constraint equivalent to the conjunction of all the constraints imposed so far. As with Prolog II, CLP replaces the unification of an atom A_i with the head of a clause A by the equation $A_i = A$. The unification succeeds if the constraint $\sigma \wedge (A_i = A)$ is solvable in the constraint system $\mathcal{C}$. This is written $\mathcal{C} \models \sigma \wedge (A_i = A)$. If A_i is a constraint c, then it is added to σ provided $\mathcal{C} \models \sigma \wedge c$. There are three transition rules:

Reduce atom

$\langle A_1, \ldots, A_i, \ldots, A_n; \sigma \rangle \rightarrow \langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1} \ldots, A_n; \hat{\sigma} \rangle$

if A_i is an atom,

$A \leftarrow B_1, \ldots, B_k$ is a renamed-apart clause of P.

$\hat{\sigma} = \sigma \wedge (A_i = A)$,

and $\hat{\sigma}$ is solvable in $\mathcal{C}$.

Reduce constraint

$\langle A_1, \ldots, A_i, \ldots, A_n; \sigma \rangle \rightarrow \langle A_1, \ldots, A_{i-1}, A_{i+1} \ldots, A_n; \hat{\sigma} \rangle$

if A_i is a constraint c,

$\hat{\sigma} = \sigma \wedge c$,

and $\hat{\sigma}$ is solvable in $\mathcal{C}$.

Fail

$\langle A_1, \ldots, A_i, \ldots, A_n; \sigma \rangle \rightarrow \langle \text{fail}; \sigma \rangle$

if A_i is an atom,

and for every renamed-apart clause $A \leftarrow B_1, \ldots, B_k$ of P,

$\sigma \wedge (A_i = A)$ is unsatisfiable in $\mathcal{C}$,

or A_i is a constraint c,

and $\sigma \wedge c$ is unsatisfiable in $\mathcal{C}$.

Again, this transition system is non-deterministic. CLP implementations backtrack to the latest choice point when failure is detected. CLP may perform *a priori* pruning when the computation fails because a constraint c is unsatisfiable in σ.

The answer is the final set of constraints projected onto the variables in the original goal. Here we see a great strength of CLP, its ability to produce symbolic output. When the exact values of variables are unknown, the interpreter is often able to print the relationship between them. For example, the answer to a CLP($\mathcal{R}$) query might be: $X > 4, Y = 3 * X$.

Often an extension to logic programming can be viewed as an instance of CLP with a suitable choice of constraint system. Standard logic programming is an instance of CLP. Its domain is the set of finite trees with syntactic equality constraints. The domain of Prolog with equality is the set of finite trees but its constraint system has a more powerful equality. E-unification is a guide for how the constraint solver works. Similarly for functional programming in logic programming. Semantic unification is an algorithm for solving the equality constraints. The domain of Prolog II is the set of rational trees. Its constraint system has equality and disequality. We have already seen CLP($\mathcal{R}$). Other CLP languages include CAL over the domain of (possibly non-linear) polynomials, CHIP over finite domains, boolean terms and rational numbers, CLP(Σ^*) over the domain of strings or regular sets, and Prolog III over finite domains, rational trees and boolean terms.

To achieve reasonable efficiency, the constraint solver is expected to handle the frequent cases quickly. Hopefully these will be the easiest constraints to solve. There will be other constraints that are considerably harder to solve. In CLP($\mathcal{R}$), one can easily ask for a solution to Fermat's last theorem, but the answer is not easy to find. One thing we can do is to delay handling hard constraints. When a hard constraint is encountered, it is not checked for satisfiability. It is put aside and execution continues. One hopes the constraint can be simplified by the addition of information as the execution progresses. The hard constraints are reconsidered when enough information has been gathered to turn them into simple constraints.

The hard constraints in CLP($\mathcal{R}$) are non-linear constraints. They are delayed until they become linear. A constraint is simplified when one of its variables is instantiated. Replacing a variable by its value can reduce the degree of a constraint. As more variables are instantiated, the constraint eventually becomes linear. For example, the goal ? $X*Y+Y=8, X+Y=5, X>1$ has the solution $X=3$, $Y=2$ but the interpreter cannot find it. The constraint $X*Y+Y=8$ is delayed because it is not linear. Now the goal ?- $X*Y+Y=8, X+Y=5, p(X)$ can be solved if $p(X)$ binds a value to X. Assume $p(X)$ binds X to 3. Then the first constraint is resumed because $3*Y+Y=8$ is now linear, and the interpreter finds the solution[JM87].

CLP is parameterized by a domain. You choose another suitable domain and you have a new language. This gives motivation to try to reuse large parts of the code between CLP implementations. We can change the implementation of CLP(X) into CLP(Y) if we replace the solver for X by a solver for Y. This will work as long as the solver is well separated from the rest of the system. The interface between the system and the solver should be small but general enough to be standardized across many solvers. In practice, all of the languages listed above were developed independently and they do not share code. Lim and Stuckey have developed a CLP shell making it easier to build CLP implementations[LS90]. In principle, you could re-implement these languages within this shell so they all share the same code.

Chapter 3

Concurrent Logic Programming

In this chapter we look at concurrent logic programming. There is a lot of parallelism available in logic programs. AND-parallelism comes from solving in parallel all the goal atoms in a goal. OR-parallelism comes from trying all relevant clauses in parallel when reducing a goal. AND-parallelism solves sub-parts of a goal in parallel, while OR-parallelism finds alternative solutions to a goal in parallel. The difficulty of implementing AND/OR-parallelism together has led to two research directions being pursued: 1) parallelizing Prolog with OR-parallelism by uncovering implicit parallelism and 2) committed-choice concurrent logic programming languages with explicit AND-parallelism. The origin of these languages can be traced to the process interpretation of van Emden and de Lucena[vEdLF82]. In their model, a goal atom is a process and a goal is a network of processes communicating through shared variables. Clark and Gregory introduced synchronization and committed-choice nondeterminism[CG81]. Synchronization is achieved by waiting for variables to be instantiated. In a committed-choice language, a goal evaluates the guards of unifying clauses and commits to one with a successful guard. There is only one solution found, there is no backtracking. A guard can contain a user predicate which calls a guard and so on. Flat languages restrict the guard to be a conjunction of goal atoms taken from a set of primitive predicates. The greater simplicity of flat languages outweighs the small loss in expressive power. The complexity of some of these issues, especially in Concurrent Prolog, is what led Saraswat to propose concurrent constraint programming and the cc framework.

3.1 Parallelism in Logic Programs

Most studies find there is ample opportunity for parallelism in logic programs, and it is easy to find, not like the imperative languages where it has to be unraveled. The evaluation of a goal forms an AND-OR tree, solving a goal implies searching through this AND-OR tree. The reduction of an atom is an OR-node, the resolvents with the candidate clauses are the children of the OR-node. A conjunctive goal is an AND-node, the atoms in the goal are the children of the AND-node. There are three widely recognized sources of parallelism in logic programs:

- *Within unification.*

  Unification is a good target for parallelization since it is performed so often during execution. Unfortunately, studies have shown that it is inherently sequential [Yas84, DKM84][Kni89, section 10 for a survey]. There is little hope of finding a parallel algorithm much faster than the sequential linear algorithm of Paterson and Wegman[PW78].

- *OR-parallelism.*

  In general, many clauses can be used to reduce a goal. The Prolog interpreter chooses the first one, and backtracks to try the other ones. We can also try all the clauses in parallel. If there are k candidate clauses to reduce a goal, k goals are produced, one for each candidate clause. All these goals are then solved in parallel. This is called OR-Parallelism, because a solution of any one of these goals is a solution of the original goal. In short, OR-parallelism explores alternative solutions to a problem in parallel.

- *AND-parallelism.*

  A goal is made up of a conjunction of goal atoms. The Prolog interpreter solves them from left to right in depth-first fashion. We can solve all the goal atoms in parallel. This is called AND-parallelism because all the goal atoms must be solved to form a solution for the entire goal. AND-parallelism solves sub-parts of the same problem in parallel.

We have to be careful with AND-OR parallelism because it can waste computation compared to Prolog's evaluation order. Prolog is working on one possible solution at a time. The children of an OR-node are looking for multiple solutions at the same time. When only one solution is needed, the first one to be found will be reported, but time and space will have been wasted searching for alternative solutions. The children of an AND-node are working on the same goal. If one of the children fails, then its siblings to the right will have done useless work. Nevertheless, searching the AND-OR tree in parallel is generally beneficial.

Executing a program in parallel involves managing a set of environments. An environment contains the set of variables created so far, if they are instantiated or not, and if so, to what values. The children of an AND-node are part of a single goal, they share the same environment inherited from the parent. When one of the children binds a variable in the parent environment, this is automatically seen by the other children. This is how communication between processes is achieved. The children of an OR-node are independent goals. They share the same environment as their parent but the bindings they make are their own and should not be seen by others. In a sense, they start with a different *copy* of the environment, so the goals do not affect each other in any way. For example, the same variable may well have conflicting values in different goals. This is how we get independent solutions.

Unfortunately, the multiple bindings created by OR-parallelism and the sharing created by AND-parallelism is difficult to implement. This resulted in two research directions being pursued[Cla90]: 1) parallelizing Prolog with OR-parallelism and 2) the creation of committed-choice concurrent logic programming languages to take advantage of AND-parallelism. In this thesis, we are mostly concerned with the latter.

3.2 Parallelizing Prolog

Some pe ple argue that writing parallel programs is difficult and a good strategy to use parallel machines is to uncover implicit parallelism from sequential programs. The responsibility for exploiting this parallelism rests on the compiler. Parallelizing compilers for Fortran have been available for some time now.

In the field of logic programming, some researchers are working on parallelizing compilers for Prolog. They exploit OR-parallelism present in sequential logic programs. They also exploit AND-parallelism as long as the AND-parallelism goals do not have variables in common. This is called independent AND-parallelism. The net effect is to ensure a variable is never shared by two concurrently executing goals. In turn, this guarantees there will be at most one binding generated for each variable, thereby eliminating the problem of multiple bindings. This greatly simplifies the implementation. OR-parallelism creates multiple bindings but these are implemented by multiple independent copies. A copy will have at most one binding. The restriction that makes the implementation practical is also its greatest weakness: OR-parallel processes cannot communicate because they cannot share variables.

AURORA is an example of an existing system. It is a prototype OR-parallel implementation of the full Prolog language for shared memory machines[LBD+88].

3.3 Concurrent Logic Programming Languages

The first process interpretation of logic programs was given by van Emden and de Lucena[vEdLF82]. In their model, a goal atom is a process and its state is represented by its arguments. AND-parallelism is used to execute processes concurrently. A goal is viewed as a network of processes communicating through shared variables.

Parallelism in logic programs is very fine-grain because processes are very short-lived. The behavior of a process depends on the clause used to ieduce it. If the clause has an empty body, the process halts. If the clause has a unit body, the process changes its state. If the clause has a conjunctive body, the process splits into several concurrent processes. For example, when the process $p()$ reduces with the clause $p() \leftarrow p1(X), p2(X)$, it will create two processes $p1(X)$ and $p2(X)$. They can communicate through the shared variable X.

The process $p1(X)$ can send a value to $p2(X)$ by instantiating X to some value. Since a logic variable can only be assigned once, it seems we can communicate at most one value through X. This is true but for example, X can be instantiated to a list cell $[v|Y]$, thereby sending the value v and creating a new shared variable Y. The processes can recurse with $p1(Y)$ and $p2(Y)$ for another round. When used this way, the shared variable is an incrementally constructed list acting like a stream of messages between the processes. More generally, X can be any shared data structure cooperatively and incrementally constructed.

Clark and Gregory, influenced by CSP (Communicating Sequential Processes), refined these ideas and introduced committed-choice non-determinism and synchronization into logic programs[CG81]. Their "Relational Language" was very influential. It is the ancestor of many concurrent logic programming languages, including Concurrent Prolog, PARLOG, GHC and Strand.

3.4 Committed-choice Non-determinism

Clause selection is different in a concurrent logic programming language. Instead of don't-know non-determinism with backtracking as in Prolog, they exhibit don't-care non-determinism also called committed-choice non-determinism. A clause $A \leftarrow G|B$ has three parts: a head A, a guard G and a body B. G is a sequence of goal atoms $G_1, \ldots, G_k$, while B is a sequence of goal atoms $B_1, \ldots, B_n$. The vertical bar is the commit operator. When a guard is empty, the clause can be simplified to $A \leftarrow B$, this is equivalent to $A \leftarrow \text{true}|B$. A unit clause is equivalent to $A \leftarrow \text{true}|\text{true}$.

A goal is a sequence $A_1, \ldots, A_m$. This goal is reduced by selecting an atom A_i in the sequence. The guards of all clauses unifying with A_i are evaluated in parallel. If there are no clauses unifying with A_i or if all guards fail, then the whole computation fails. Otherwise, there is at least one clause with a successful guard. One such clause is chosen and the goal commits to it. The evaluation of the other guards is aborted.

There is no backtracking, the other clauses will not be tried later. Only one solution is generated. A guard has to be written in such a way that it is successful only when its clause is certain to generate a solution if one exists, otherwise, the goal may commit to the wrong clause and the solution will be missed. A computation may fail when in fact the goal has solutions.

A guard can contain a user predicate, this predicate may invoke new guards and so on. The guards may be nested arbitrarily deep. The computation forms an AND-OR tree. A guard is an AND-node and clause selection for a user predicate in a guard is an OR-node.

Only the guard of the chosen clause may have side-effects. The atomicity of clause selection is ensured by the requirement that at most one binding for each shared variable is ever generated. A clause cannot instantiate the variables in the goal until the goal has committed to this clause. A clause head cannot instantiate the variables in the goal because those bindings will have to be removed if the goal commits to another clause. For the same reason, a guard cannot instantiate variables in the goal.

What happens if a clause tries to instantiate variables in the goal before the goal has committed varies among languages. In PARLOG, a compile-time check guarantees this will never happen, but this is undecidable in general. The check can only be an approximation of the full test. In practice, PARLOG relies on the programmer to avoid the situation. In GHC, a guard will suspend when it tries to instantiate a variable in the goal. The clause will be considered again when the variable has been instantiated by some concurrently executing process. Of course, the clause will be reconsidered only if the goal has not committed to another clause before then.

In Concurrent Prolog, the guards are evaluated in their own environment, similar to OR-parallelism. The head and the guard are allowed to instantiate goal variables but this is not visible outside the guard. A clause with a successful guard is chosen for commitment. The environment of the chosen clause is merged with the goal to form the environment of the resolvent. The goal can commit if all new bindings of the resolvent environment can be made with a single multiple assignment. If a binding fails, then none of the bindings are made. The bindings are made visible after the whole assignment succeeds. If they were made visible earlier, then the assignment could fail and these bindings would have to be retracted. Other concurrent processes could see these bindings before they were retracted and clause selection would not be atomic.

The atomic operation in PARLOG, GHC and Strand is binding a single variable. Bindings are made as unification proceeds, but unification itself is not atomic. In Concurrent Prolog, binding a set of variables is the atomic operation. This is needed to merge the environment of the goal with the environment of the chosen guard.

The difficulty of supporting user predicates in guards has directed attention towards the flat variants of the concurrent logic programming languages. In a flat language, only a set of primitive predicates can appear in a guard. For the languages we mentioned, this includes mainly unification and arithmetic tests. A flat language is easier to implement because the computation does not form an AND-OR tree, instead it forms a flat collection of processes.

For example, the suspension rule in GHC is difficult to implement because the interpreter must know at which level of the tree the variables belong to[Tay88]. There is no need for this in Flat Guarded Horn Clauses (FGHC). The primitives in FGHC guards are test-only automatically suspending predicates, i.e., their arguments must be instantiated otherwise they suspend. This together with the flatness of the tree guarantees a guard will never instantiate a goal variable. There is no need for an elaborate runtime check[Cla90].

Most implementations of concurrent logic programming languages are for the flat subset. It seems the greater simplicity of flat languages outweighs the small loss in expressive power.

3.5 Synchronization

Synchronization is achieved by waiting for variables to be instantiated. For example, the reader of a stream will suspend until the shared variable is instantiated announcing the arrival of the next element. How synchronization is accomplished differs among languages. PARLOG uses mode declarations. For each argument of a predicate, the mode declaration specifies if it is input or output. A mode declaration applies to a whole procedure, i.e., all clauses with the same predicate name and same arity. A clause suspends if it tries to instantiate an input argument, so processes synchronize by waiting for input. In GHC, a guard suspends if it tries to bind goal variables.

In Concurrent Prolog, synchronization comes from read-only unification. Variables come in two types: $X?$ is the read-only variable associated with the writable variable X. Unification of two terms suspends if an attempt is made to instantiate a read-only variable. The unification may resume when the variable is instantiated by binding the writable variable. The read-only attribute is a dynamic property determined at run-time. The property is transitive: if a writable variable X is bound to a read-only variable $Y?$, then X becomes read-only. Read-only unification encourages the use of unification when matching is intended. This is a source of bugs. The implementation of Concurrent Prolog requires distributed atomic read-only unification. This is very difficult to implement and not very efficient.

The problems in Concurrent Prolog with read-only unification and the atomic merge of environments led Saraswat to search for a simpler theory. The result was the concurrent constraint programming framework discussed in the next chapter and implemented in Taskell.

Chapter 4

Concurrent Constraint Programming

This chapter is based on the book "*Concurrent Constraint Programming*" by Saraswat [Sar89]. Concurrent constraint programming is the synthesis of Constraint Logic Programming and the committed-choice concurrent logic programming languages. From CLP, it adopts computing with constraints and programs in clausal form. From Concurrent Prolog, it adopts parallelism and guarded Horn clauses. To this, it adds parallelism in the constraint solver and synchronization based on what can be inferred from the constraints.

Concurrent constraint programming replaces the usual view of memory by the store, a constraint on the variables that may only partially specify them. It also replaces the notion of reading and writing memory by the primitives ask and tell. The cc framework is parameterized by a constraint system. Different choices of constraint system gives various cc instances. Processes in the framework are called agents. The framework defines a process language with a set of agent combinators. A cc instance may retain only a subset of the available combinators. In most instances, an agent may engage in a primitive operation or it can split into multiple agents, it can make a dependent choice, it can hide a variable from the outside environment or it can make a procedure call. CLP, most concurrent logic programming languages, Janus, GDCC and Taskell are all instances of cc.

4.1 The Ask and Tell Primitives

Constraint programming cannot share the usual view of memory found in most languages. Traditionally, the memory can be thought of as a valuation, i.e., a mapping from variables to values. Reading memory means consulting this mapping for the value of a variable and writing memory modifies the mapping for subsequent reads. The concepts of reading and writing become incoherent in the presence of constraints. The constraints describe the relationship between variables but not necessarily their values. A variable may be constrained but not determined by all the constraints posted so far. This variable could take one of possibly many values. No single value returned by the read operation can capture this multiplicity because read only works on ground values. Assigning a value to a variable may fix the value of other variables through the constraints that were imposed on them. This propagation of constraints is lacking in the write operation because write effects only one variable.

The cc framework replaces memory by the store. The store is a representation of the conjunction of the constraints imposed so far. It is always kept consistent to ensure the feasibility of a solution. As new constraints are imposed, they refine the values of variables. The read and write memory operations are replaced by the primitives blocking ask and atomic tell respectively.

Let c be some constraint and A be an agent. Processes in cc are called agents. A blocking ask has the form: ask(c) $\rightarrow$ A. Its behavior is as follows: if c is entailed by the store then carry out the process A, otherwise suspend. A blocking ask may resume only when other agents have added enough information to infer c from the store. This suspension mechanism is the source of synchronization in cc. Ask is a stable operation: if the ask may proceed in the current store, it could also proceed in any extensions of the store (because information is never retracted from the store). Ask behaves like a guard in concurrent logic programming languages. In practice, it is used as a conditional.

Let c be some constraint, σ be the current store and σ' be the new store. The behavior of tell(c) is as follows: if $\sigma \wedge c$ is consistent then atomically let $\sigma' = \sigma \wedge c$, i.e., augment the store with this new constraint, if $\sigma \wedge c$ is inconsistent then abort the whole program. Tell adds information to the store by imposing new constraints. If a tell makes the store inconsistent, the whole program is aborted. In an inconsistent store, all asks are entailed. This leads to uncontrolled behavior because all asks may proceed. This situation is considered a programming error. The run-time system should abort the program and signal the error.

Like CLP, cc is parameterized by a constraint system $\mathcal{C}$. From $\mathcal{C}$ we require the possibility to talk about partial information through primitive and possibly compound constraints. We require a notion of consistency to know which constraints can hold of the same object, and finally, a notion of entailment to know what we can infer from a set of constraints.

The only explicit constraints in CLP are primitive constraints. Compound constraints are built-up by the evaluation procedure. For example, conjunctions are formed sequentially by the comma operator, and disjunctions are explored one branch at a time through alternative clauses and backtracking. In cc, the constraints may be either primitive or compound.

We will use $\mathcal{C} \models c$, read "$\mathcal{C}$ entails c" to mean c is true in the constraint system $\mathcal{C}$. For a store σ, $(\exists)\sigma$ is the existential closure of σ σ over all variables. The following definitions will be used in the description of ask and tell below:

σ answers c	if $\mathcal{C} \models \sigma \Rightarrow c$ i.e., when the store contains enough information to entail c.
σ suspends c	if $\mathcal{C} \models (\exists)(\sigma \wedge c)$ and $\mathcal{C} \models (\exists)(\sigma \wedge \neg c)$ i.e., when the store does not contain enough information to entail c.
σ accepts c	if $\mathcal{C} \models (\exists)(\sigma \wedge c)$ i.e., when c is consistent with the store.
σ rejects c	if $\mathcal{C} \not\models (\exists)(\sigma \wedge c)$ i.e., when c is inconsistent with the store.

We can now give a definition for blocking ask: ask(c) $\rightarrow A$

when σ answers c	behave as A
when σ suspends c	suspend this agent until σ answers c and then behave as A

and the definition of atomic tell: tell(c)

when σ accepts c	$\sigma' = \sigma \wedge c$ atomically augment the store with c and terminate this agent
when σ rejects c	abort the whole program

Ask and tell primitives are messages to the constraint solver in the store. There is little contention in the store because the solver can be highly parallel. There is little need for locks because there is only a benign form of change: adding consistent information. In this setting, it is easy to solve multiple asks in parallel, and it is also possible to have multiple tells solved in parallel.

Saraswat defines other operations like eventual tell, inform and check. We motivate eventual tell in the section on implementation considerations below. Inform is like tell but it succeeds only if some new information is added to the store. Check tests to see if a constraint is consistent with the store without actually adding the constraint. Inform and check give some of the power of the var predicate in Prolog.

4.2 The Process Language

Processes in cc are called agents, multiple agents execute in parallel. The framework defines a process language with a set of agent combinators (such as $\|$, $+$ below). A cc instance may retain only a subset of the available combinators. Choosing more combinators makes the language more powerful, but this increases the complexity of the implementation. We describe an instance by its choice of constraint system and its subset of agent combinators.

Agents do not communicate directly with each other. There is no notion of sending a message to a specific agent as there is in CSP. Instead agents communicate through shared variables in the store. Synchronization arises because ask behaves like a blocking receive waiting for the information tell will send.

An agent may engage in a primitive operation, it can split into multiple agents, it can make a dependent choice, it can hide a variable from the outside environment or it can make a procedure call. The syntax and the behavior of these combinators are introduced below.

The effect of $A_1 \parallel A_2$ is to run A_1 and A_2 in parallel.

Choices are made with $\text{ask}(c_1) \rightarrow A_1 + \cdots + \text{ask}(c_n) \rightarrow A_n$. In this context, $\text{ask}(c_i)$ is called a guard. A guard is open if its constraint is entailed by the store. The choice operator has the following effect: if all guards suspend then it suspends, otherwise it non-deterministically chooses an open guard and it behaves like its guarded agent. For example, if $\textbf{guard}_2$ is entailed, then the choice operator may choose to behave like A_2. The choice operator is needed when the guards are not mutually disjoint, otherwise the component asks could be run in parallel with the $\parallel$ combinator.

Saraswat also defines tell guards. The agent $\text{tell}(c_1) \rightarrow A_1 + \cdots + \text{tell}(c_n) \rightarrow A_n$ will behave as A_i in the store $\sigma \wedge c_i$ only if c_i is consistent with σ and i was the chosen branch. Tell guards that are not chosen do not have any effect on the store. Tell guards may compete to bind a variable. For example, in a store where X is unbound, the agent $\text{tell}(X = 1) \rightarrow A_1 + \text{tell}(X = 2) \rightarrow A_2$ will behave as A_1 if X was bound to 1, or it will behave as A_2 if X was bound to 2. A choice operator may have mixed ask and tell guards.

The effect of the existential closure $\exists x.A$ is to behave as A with the variable x local to A. It hides x from other agents by creating a new variable x local to A.

An agent makes a procedure call with the syntax $p(X)$, where X is a vector of values or variables. Procedures are only defined once, choices have to be programmed explicitly with the choice operator. Recursion is allowed.

A program is a collection of procedures and an initial agent. A procedure definition looks like this:

proc $p(x_1, \ldots, x_k)$

⟨inner definitions⟩

A

where A is an agent. A procedure may have nested procedures as in a nested-block language.

A computation may go on forever or terminate. If a computation terminates it can either fail with an inconsistent store in which case the program is erroneous, or it terminates normally in which case the result of the computation is the final store. A program may terminate if all agents have terminated or if all remaining agents are suspended. Note that deadlock is not necessarily an error.

The cc framework is not merely a committed-choice language. There are other combinators like the OR-parallel search operator which is possibly implemented through backtracking. With this operator, we see that CLP is a special case of cc. One needs to rewrite the clause selection as an explicit OR-parallel search, there is no parallel operator since everything is sequential and the same constraint system is used for CLP and cc. The flat concurrent logic programming languages are also a special case of cc. Clause selection must be written explicitly with the choice operator. The constraint system C is the set of finite trees with equality constraints.

It is possible to give a transition system for cc. Saraswat gives such a transition system in his thesis[Sar89]. The paper by Saraswat and Rinard gives the semantics of cc as a reactive congruence through bisimulation[SR90]. The subsequent paper by Saraswat, Rinard and Panangaden gives the semantics of cc agents as closure operators[SRP91].

4.3 Implementation Considerations

In this section we examine some issues raised by the implementation. Eventual tell is motivated by the desire to spread the store in a distributed memory architecture. A weak entailment relation is needed to simplify the solver. Each Agent may answer its own query so the store need not be an active entity. Suspended agents may be aborted if we can find out they will be suspended forever. Finally, the store may be garbage collected to reuse the storage.

Eventual Tell. To implement atomic tell efficiently, we need shared memory. Shared-memory machines do exist but they do not scale as well as distributed memory machines. Two possibilities for implementing cc on a distributed memory machine is to replicate the store on many nodes, or to split the store onto many nodes. These open many interesting avenues but they are not easy to implement.

Another possible solution is to use a local store on each node and a global store. Agents tell constraints in the local store, possibly checking consistency locally but delaying the consistency check in the global store. Periodically the local store is combined with the global store and a full consistency check is performed. The store is now responsible for consistency checking. We can define a new primitive called eventual tell. The call etell(c) adds c to the local store consistently and terminates. The constraint c will eventually migrate to the global store and be checked for global consistency at that time.

Ask is still a stable operation, an agent may simply wait longer because the information it is waiting for is sitting in a local store on another node. The information will eventually propagate to the global store and this will resume the ask. The operation etell(c) is not atomic, for example the agent etell($X = 1$)||etell($X = 2$) may succeed even though the values of X are inconsistent. Eventually, the bindings for X will migrate to the global store and the program will abort because the inconsistency will be detected at that stage. This only happens with erroneous programs.

Deep guards in concurrent logic programming languages are closely related to eventual tell. When a guard is evaluated, the bindings it makes are not seen by the other goals. This is like making bindings in a local store. When the clause commits, the bindings of the guard are made available to the other goals. This is like migrating constraints from the local store to the global store. The local store of the guards that were not chosen will never migrate to the global store. The evaluation of a guard begins in a new local store so it does not see the effect of an old unsuccessful guard.

Weak Entailment Relations. Some constraints are hard to solve. We saw how the CLP implementation puts the hard constraints on a suspension queue. In cc the same effect is achieved by weakening the entailment relation. We may wish to wait for more information before we infer a constraint c even when the store logically entails c. For example, an entailment relation might be too weak to infer that $X = 2$ from the store $X * X = 4$. The agent ask($X = 2$) $\rightarrow$ A will be suspended even though the store logically entails the constraint. Ask remains a stable operation, but the suspended asks may have to wait longer. The advantage is a greatly simplified implementation of the solver.

We can also simplify the tell operations. We may wish to suspend a tell until an implied ask is entailed. In the implementation of naive arithmetic, all arguments to functions must be ground before the function can be applied. This can be done by introducing implicit asks in constraints. For example, tell($X = Y + Z$) implicitly asks that Y and Z both be ground. This tell is equivalent to ask(integer(Y) $\wedge$ integer(Z)) $\rightarrow$ tell($X = Y + Z$). Implied asks can be used to test the types of operands as above. Previously, a tell was either consistent and it terminated, or inconsistent and the program failed. Now a tell may suspend because one of its implied asks suspends.

Another way to simplify the constraint system is to specialize the constraints to either ask or tell but not both. An ask-only constraint can be used in an ask but not in a tell. Similarly a tell-only constraint can be used in a tell but not in an ask. Typically, value constructors are tell-only while type recognizers and component selectors are ask-only.

Aborting infinitely suspended agents. If the store entails $\neg c$, then the constraint c will never be entailed and the agent ask(c) $\rightarrow$ A will suspend forever. In some constraint systems it is possible to determine that $\neg c$ is entailed when checking to see if the store answers c. In that case, the implementation may wish to abort the agent to save storage because this is observationally indistinguishable to an infinitely suspended agent.

Agents may answer their own query. An ask checks if its constraint is entailed by sending a query to the store and waiting for the reply. The solver will eventually get around to this request, and try to solve it. If the constraint is entailed, the solver will reply with a message to resume the ask. If it realizes the negative of the constraint is entailed, it will reply with a message to abort the ask. Otherwise, the solver remembers the request (or a simplification of it). These requests are reconsidered when more information is added to the store.

The agent is blocked waiting for the reply throughout the time the solver is working on its behalf. While the agent sits idle, the solver had to fork a worker thread to solve this constraint. It seems more economical if the solver could borrow the thread of control of the agent instead of forking a new thread. The store is no longer an active entity since the agents answer their queries for themselves. This saves on the number of processes as well as two interprocess communications because the request need not be sent nor replied. This is a common technique in remote procedure call implementations.

Garbage Collection. As defined, it seems the store will keep on increasing by the addition of new constraints. The solution is the same one found in LISP systems The store should be garbage collected. Portions of the store guaranteed not to be needed anymore can be reused. For example, the constraint $X < 5$ may be discarded if it is known that $X < 2$. Each constraint system will handle this differently. For some constraint systems, this may not be obvious at all.

4.4 Implementations of cc

There are many implementations of cc languages. Most were not conceived as instances of cc. The cc framework was born out of the attempt to unify the many variants of concurrent logic programming languages. Understandably, most of these languages can be seen as instances of cc. One notable exception is Concurrent Prolog. The subtleties of read-only unification and merging the environments cannot be modeled in the framework.

Janus is the first language specially designed as an instance of cc[SKL90, SKL89]. Janus has a clausal syntax closer to GHC than the cc syntax we defined here. The constraint system of Janus is designed so a constraint can never fail. The basic data types are bags, or multi-sets based on the theory of non well-founded sets, rational (infinite) trees and updatable arrays. Janus has the two-occurrence restriction originally found in Doc[Hir86]: a variable can occur at most twice in a clause. Furthermore, the occurrences are classified as either a teller or an asker. A teller has write capability to a variable; an asker has read capability. Capabilities are consumed after only one use but they may be passed as arguments without consuming them. Askers and tellers are created in pairs so an asker knows there is a teller that may possibly instantiate the variable, and the teller knows there is an asker ready to read the value it has produced. The two-occurrence restriction seems very drastic, nevertheless, most logic programming techniques can be used in Janus. This includes producer-consumer interactions, short circuits, incomplete messages and messages into the future. Bags are used to get many to one communication. This makes it possible to write servers respecting the two-occurrence restriction. Janus has a sequential implementation running on top of SICStus Prolog[Deb91].

The domain of values of the language 'Guarded Definite Clauses with Constraints' or GDCC is the set of rational polynomials. Its solver is a parallel implementation of the Buchberger Algorithm. To the author's knowledge, GDCC is the first parallel implementation of a language specifically designed as an instance of cc[Haw91]. The language Taskell described in the following chapter is another cc instance implemented in parallel.

Chapter 5

The Programming Language Taskell

This chapter describes the programming language Taskell, a member of the cc family. Each member of cc is characterized by its constraint system and its set of agent combinators. The constraint system of Taskell is the set of finite trees with equality. Taskell also has naive arithmetic on integers. The agent combinators are procedure calls, `||` for parallel execution, `new` for existential closure and `choice` for selective behavior. The `main` procedure declares the variables that are printed at the end of the computation for the results. The syntax of Taskell is given in figure 5.1, it closely resembles the syntax used in chapter 4. After the language is described, the last section illustrates its use with a set of sample programs demonstrating list processing capabilities, tightly-coupled producer-consumer relationships and non-deterministic processing.

5.1 Definition of Taskell

The syntax of Taskell is given in figure 5.1. Symbols in the syntax have the following meaning: a word between ⟨*angle-brackets*⟩ is a syntactic category, the character | means 'or', the character ε is the empty string, the ellipsis in ⟨*cat*⟩ *sep* … *sep* ⟨*cat*⟩ means either the empty string ε, or ⟨*cat*⟩ , or else two or more ⟨*cat*⟩ separated by *sep*. Other characters are in `typewriter` font and appear explicitly in the program.

One of our design principle was to minimize the use of special characters, therefore we chose to spell the name of operators explicitly instead of using symbols. Non-alphanumeric characters are used for punctuation only. We feel this makes Taskell's syntax easy to learn and remember. The following paragraphs discusses the rationale for our design.

The following keywords are reserved: **and**, **ask**, **begin**, `choice`, **end**, **in**, **isint**, **istree**, **new**, `or`, `proc`, **tell**.

A program begins in the procedure called **main**, so the body of **main** is the initial agent. One cannot send arguments to **main** even though it has parameters. These variables contain the results of the computation at termination and are printed at the end of the execution. Taskell does not have other I/O mechanisms. A procedure can have internal procedures as in a nested-block language. Free variables in the body are resolved with lexical scoping. A procedure body is a single agent, possibly built from the parallel combinator ||. The scope of the parallel combinator is delimited by a pair of enclosing braces. The braces are necessary to remove any ambiguity in the scope of **new**. It is possible to enclose a single agent in braces to make the scoping more explicit. It is legal to always put braces after an **ask** or a **new** even though they are not necessary in all cases. The definition of a procedure with no arguments needs the parentheses, e.g., `proc fred()`, which is called with an empty list of arguments.

The symbol || approximates the parallel symbol ‖. The symbol '->' in blocking ask is meant to point to the agent executed *after* the constraint is entailed. The guards in a choice are separated by the symbol '+'. The plus sign denotes the union of possible choices available to the agent. A guard looks like a regular ask because if the guards are all disjoint, the choice operator is equivalent to all these asks running in parallel. We chose to use => in guards instead of -> to remove some of the confusion created by the two uses of **ask**. Using => also helps error recovery in the parser. Taskell does not have tell guards, but these could be a future extension.

The values in the domain are integers and finite trees. A finite tree is made up of a root name and a list of children separated by commas. The root of the tree

```
⟨program⟩         ≡ proc main ( ⟨var⟩ , ... , ⟨var⟩ ) ⟨procs⟩ begin ⟨agent⟩ end
⟨procs⟩           ≡ ⟨proc⟩ ⟨procs⟩ | ε
⟨proc⟩            ≡ proc ⟨name⟩ ( ⟨var⟩ , ... , ⟨var⟩ ) ⟨procs⟩ begin ⟨agent⟩ end
⟨agent⟩           ≡ ⟨ask⟩ | ⟨tell⟩ | ⟨new⟩ | ⟨par⟩ | ⟨call⟩ | ⟨choice⟩
⟨ask⟩             ≡ ask( ⟨ask-constraint⟩ ) -> ⟨agent⟩
⟨tell⟩            ≡ tell( ⟨tell-constraint⟩ )
⟨new⟩             ≡ new ⟨var⟩ , ... , ⟨var⟩ in ⟨agent⟩
⟨par⟩             ≡ { ⟨agent⟩ || ··· || ⟨agent⟩ }
⟨call⟩            ≡ ⟨name⟩ ( ⟨value⟩ , ... , ⟨value⟩ )
⟨choice⟩          ≡ choice { ⟨guard⟩ + ··· + ⟨guard⟩ }
⟨guard⟩           ≡ ask( ⟨ask-constraint⟩ ) => ⟨agent⟩
⟨ask-constraint⟩  ≡ ( ⟨ask-constraint⟩ ) |
                    ⟨ask-constraint⟩ and ⟨ask-constraint⟩ |
                    ⟨ask-constraint⟩ or ⟨ask-constraint⟩ |
                    isint( ⟨value⟩ ) |
                    istree( ⟨name⟩ , ⟨integer⟩ , ⟨value⟩ ) |
                    ⟨value⟩ = ⟨value⟩ |
                    ⟨value⟩ > ⟨value⟩ | ⟨value⟩ >= ⟨value⟩ |
                    ⟨value⟩ < ⟨value⟩ | ⟨value⟩ <= ⟨value⟩
⟨tell-constraint⟩ ≡ ( ⟨tell-constraint⟩ ) |
                    ⟨tell-constraint⟩ and ⟨tell-constraint⟩ |
                    ⟨value⟩ = ⟨value⟩
⟨value⟩           ≡ ( ⟨value⟩ ) | ⟨var⟩ | ⟨integer⟩ | ⟨tree⟩ |
                    ⟨value⟩ + ⟨value⟩ | ⟨value⟩ - ⟨value⟩ |
                    ⟨value⟩ * ⟨value⟩ | ⟨value⟩ / ⟨value⟩
⟨tree⟩            ≡ ⟨name⟩ | ⟨name⟩ ( ⟨value⟩ , ... , ⟨value⟩ )
⟨var⟩             ≡ ⟨name⟩
⟨integer⟩         ≡ ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩
⟨name⟩            ≡ ⟨alpha⟩ | ⟨alpha⟩ ⟨alphanum⟩
```

Figure 5.1: The syntax of Taskell.

`fred(10, X)` is `fred`, its first child is the integer 10, and the second child is the variable `X`. A tree with no arguments is written without parentheses, for example `fred` is a nullary tree. Inside a constraint, a ⟨*name*⟩ is either a nullary tree or a variable. The compiler can determine which one because variables must be declared in a parameter list or in a `new`. By convention, variables start with a capital letter, but this is not enforced.

Constraints are always enclosed in parentheses. `And` has higher precedence than `or`. The multiplicative operators (`*` and `/`) have higher precedence than the additive ones (`+` and `-`). All arithmetic operators are left associative. One can use parentheses to override the default precedence. The relational operators are `<`, `<=`, `>` and `>=`. They take two arguments and they cannot form chains: `1 < X < 10` must be written `1 < X and X < 10`.

Taskell has equality constraints between two values, for example `X = Y`, `X = 1` or `2 = 3`. Note that imposing the last constraint would make the store inconsistent. The ask-only constraint `isint(X)` can be used to ensure that `X` is an integer. The argument can be any value but it will be most useful if `X` is a variable. This constraint suspends until its argument is instantiated.

Taskell implements naive arithmetic through binary functions. These functions take two integer arguments and return an integer. There is an implied ask on each argument to guarantee they are both integer. For example, the agent `tell(X = Y * Z)` is equivalent to the agent `ask(isint(Y)) -> ask(isint(Z)) -> tell(X = Y * Z)`. The implied asks ensure that arithmetic is always performed on ground integer values. The relational operators expect their arguments to be integers. They have similar implied asks.

The ask-only constraint `istree(r, n, t)` is entailed only if `t` is a tree with root `r` and its arity is `n`. This is a restricted form of existential quantification equivalent to $\exists A_1, \ldots, A_n\ t = r(A_1, \ldots, A_n)$. Taskell does not have this form of existential quantification. `Istree` contains the arity explicitly to distinguish between trees with the same root name but different arity. Note that the arity must be an integer constant, it cannot be an integer valued expression.

It is possible to create cyclic structures in Taskell. These can create infinite loops in the run-time system. The following agent demonstrates the problem:

```
new X, Y, Z in
     {tell(X=f(X)) || tell(Y=f(Y)) || ask(X=Y) -> tell(Z=1)}
```

The first two tells succeeds because X and Y are unbound, but the ask will loop trying to check if X=Y is entailed.

Taskell is obviously an instance of cc. It can also be viewed as a concurrent logic programming language. The language closest to it is Flat Guarded Horn Clauses Taskell is flat in the sense that guards can only be a conjunction of system constraints, there are no procedure calls in guards. A disjunction in a guard can be viewed as a short-hand for two guards with the same guarded agent. Taskell's unification is not atomic, bindings are made as the unification proceeds. This distinction is not as important as in FGHC because it should only be visible in cases leading to an inconsistent store. These programs are considered erroneous and will be aborted.

5.2 Examples of Taskell Programs

This section gives a feel for programming in Taskell by listing some sample programs. The examples include an append procedure, a merge of two lists and the construction of an admissible list.

The first program demonstrates simple list handling capabilities, it computes the result of appending the list (1) to the list (2 3). The answer is Ans = cons(1, cons(2, cons(3, nil))) which is the full form of the list (1 2 3). Append tests its first argument, if it is nil it stops immediately, if it is a cons cell, it names its car and cdr X1 and X2, puts X1 in the result and calls itself recursively.

```
(* A program appending the list(1) and (2 3) *)

proc main (Ans)

      proc append (X, Y, Z)
```

```
        begin {
              ask(X=nil) -> tell(Y=Z)
           || ask(istree(cons,2,X)) ->
                 new X1, X2, Z1 in {
                       tell(X=cons(X1,X2))
                    || tell(Z=cons(X1,Z1))
                    || append(X2,Y,Z1)
                 }
        }
        end

  begin
        append(cons(1,nil), cons(2,cons(3,nil)), Ans)
  end
```

The second program is a demonstration of tightly coupled interaction between agents, it is inspired by a similar program in [Sar89]. The problem is to compute an admissible list, i.e., a list made up of pairs (X,Y) represented by the tree cons3(X,Y,Z) where Z is the rest of the list, with the following properties. The second element of a pair is twice the first element: $Y = 2 * X$. The first element is three times the second element of the previous pair: $X = 3 * \mathtt{previousY}$. The double procedure looks at one pair at a time and sets the value of the second element to twice the value of the first element. It does that for all pairs in the list. The triple procedure inspects a pair and expands the list if the element has not exceeded a threshold (30000). The main procedure forks three agents, one to guarantee the doubling requirement, one to guarantee the tripling requirement and one to initialize the list. This program will continuously switch between the double and the triple agents. The double agent needs the first element before it can compute the second element, and triple needs the second element before it can decide to expand the list or not. The answer is L = cons3(1, 2, cons3(6, 12, cons3(36, 72, cons3(216, 432, cons3(1296, 2592, cons3(7776, 15552, cons3(46656, 93312, nil)))))))

```
(* A program to compute an admissible list *)

proc main (L)
```

```
      proc double (L)
      begin {
         ask(istree(cons3,3,L)) -> {
            new X, Y, L1 in {
                  tell(L=cons3(X,Y,L1))
               || tell(Y=2*X)
               || double(L1)
            }
         }
      }
      end

      proc triple (L)
      begin {
         new X,Y,L1 in {
               tell(L=cons3(X,Y,L1))
            || ask(Y<30000) ->
                  new Y1,Z,L2 in  {
                        tell(L1=cons3(Y1,Z,L2))
                     || tell(Y1=3*Y)
                     || triple(L1)
                  }
            || ask(Y>=30000) ->
                  tell(L1=nil)
         }
      }
      end

begin {
         double(L)
      || triple(L)
      || new S,U in tell(L=cons3(1,S,U))
}
end
```

The following program is the standard example for the need of `choice`, it tests a merge procedure by merging a list of ones and a list of twos. The producer builds a list of **Max** length by repeating the value in **Id**. If the desired length is attained it stops the list, otherwise it adds a copy of **Id** to the list and calls itself recursively. Once the two producers in **main** have terminated, **L1** will be a list of twenty 1's, and **L2** will be a list of thirty 2's. The merge procedure can stop as soon as one of the list is exhausted. Otherwise it non-deterministically picks one list with an available cons cell, adds the cell to the result and calls itself recursively. Merge flips its argument in an attempt to be more fair between the two lists. The producers construct the lists as they are merged, but **merge** will never get ahead of the producer since the **istree** guard will suspend until the cons cell is available.

```
(* A program to merge a list of ones with a list of twos *)

proc main (L)

    proc producer (N, Max, Id, L)
    begin {
       new L1, NN in
           choice {
                 ask (N>Max) => tell (L=nil)
               + ask (N<=Max) => {
                        tell (L = cons(Id, L1))
                     || tell (NN = N+1)
                     || producer (NN, Max, Id, L1)
                 }
           }
    }
    end


    proc merge (L1, L2, L3)
    begin {
       new L11, L22, L33, Item in
           choice {
                 ask (L1 = nil) => tell (L3 = L2)
               + ask (L2 = nil) => tell (L3 = L1)
               + ask (istree(cons,2,L1)) => {
                        tell (L1 = cons(Item,L11))
                        || tell (L3 = cons(Item, L33))
                        || merge (L2, L11, L33)
                    }
                  + ask (istree(cons,2,L2)) => {
                           tell (L2 = cons(Item,L22))
                        || tell (L3 = cons(Item, L33))
                        || merge (L22, L1, L33)
                    }
                }
            }
            end

begin
        new L1, L2 in {
               producer (1, 20, 1, L1)
            || producer (1, 30, 2, L2)
            || merge (L1, L2, L)
        }
end
```

Chapter 6

Implementation

The implementation of Taskell has to address two issues: how is the specific constraint system implemented and how are the process combinators implemented?

The implementation of Taskell is written in Concurrent ML, a language based on message passing between concurrent sequential processes. The Taskell compiler takes a source file and produces a Concurrent ML program in another file. The compilation process is described by the mapping from the source language to the Concurrent ML intermediate code.

Many issues in this chapter depends on the specific constraint system of Taskell. For example, since the constraint system of Taskell is equational, equivalence classes play an important role. Agents may suspend for two reasons: they may wait for a variable to be instantiated, or for two variables to be merged in the same equivalence class.

The naive arithmetic operators of Taskell are value returning functions with implied asks on their arguments. This unifies the treatment of arithmetic in asks and tells.

Our implementation solves both asks and tells in parallel. Because of this, variables are implemented as processes with a protocol between agents and between other variables. The protocol is motivated and described. A variable process implements a set of variables that have been equated, we call this set an equivalence class. Finally, we give the unification algorithm used by agents to solve ask and tell constraints.

6.1 Implementation Issues

Amount of parallelism. In choosing the architecture of the implementation, the amount of parallelism is a key design decision. The agents execute in parallel and they all access the shared store. Clearly, parallelism in the store operations is very important. We will concentrate on ask and tell since other store operators are just variations of these. There are different options depending on how many asks and tells we solve in parallel.

The first option is to serialize all asks and tells into one stream of requests. Inside the store is a process that reads the requests one by one and executes them in order. Some requests can be solved immediately, others have to wait in queues. Solving a request may reactivate other requests which are then dequeued and solved. The whole request is the atomic action except that queued requests may define a smaller atomic action. Serialization may seem strange for a concurrent language but it is the most sensible thing to do in a simulation written in a sequential language. For example, QDJanus fits in this category. Serialization is also useful for debugging purposes.

The next option is to run agents in parallel and solve multiple asks concurrently with a single tell. Tell requests are merged into a single stream to be read by a process in the store as above. For an atomic tell, an agent must block until its tell request is processed by the teller. Eventual tell is achieved if the agent continues immediately after inserting its tell request into the stream. The local store is the stream of requests and consistency is checked later by the teller when it reads that request from the stream. Ask requests are conceptually solved by the store, but since an asker is blocked until the constraint is entailed, the store can borrow the asker's thread of control to solve the request. The end result is that agents solve their own ask. An ask checks the entailment of a constraint, it does not change the store. If it needs to modify the store in order to carry out an inference, this can be done in a local environment which will be discarded later. We expect an ask will not disturb the execution of another ask running concurrently, hence an asker does not have to worry about the execution of other asks. This leaves the asker to worry about the execution of the tell. The tell may change the store as the asker is trying to solve its constraint. You need a form of atomicity in the actions of the tell so that askers can account for a changing store.

Atomicity. The most interesting approach is to solve both asks and tells in parallel. This requires locks as can be seen by the following example. Let A_1 be the agent `tell(X=1)` and let A_2 be the agent `tell(X=2)`. A trace of the agent $A_1 \parallel A_2$ might be:

A_1: determine the value of `X` is `Unbound`

A_2: determine the value of `X` is `Unbound`

A_1: set `X=1`

A_2: set `X=2`

The store should be inconsistent but this was not detected. One must be careful with locks because it is easy to fall into deadlocks. A lot of the complexity in the solver involves atomicity issues. The section on the implementation of variables will discuss many small problems encountered in trying to solve asks and tells in parallel. Many details concern the atomicity of operations on variables.

Suspension mechanisms and consistency. The following discussion is only meaningful for equality constraint systems. Agents may suspend for two kinds of equalities to be entailed. The agent `ask(X=2)` needs the value of `X` to check it against the value 2. Compare this to the agent `ask(X=Y)`. The constraint `X=Y` may be entailed if `X` and `Y` have no value but are bound to each other. That is why the solver separates the suspension requests in two categories:

- An *instantiation suspending agent* for the variable `X` suspends until `X` is instantiated because the agent needs the value of `X`.

- An *equation suspending agent* for the variable `X` suspends until `X` is equated to another variable `Y`. This will happen if the equivalence class of `X` is merged with the equivalence class of `Y`. The value of `X` can be equal to the value of `Y` even if the class of `X` is not equal to the class of `Y`, so the equation suspending agents for

X are resumed when X is instantiated. An agent resumed because the variable X was instantiated will probably choose to become an instantiation suspending agent for Y.

There are some alternatives when implementing ask. In the first alternative, the constraint is checked for consistency in depth-first fashion. A sub-problem is solved completely before another sub-problem is attacked. The asker may suspend waiting for a variable to be instantiated or waiting for two variables to be equated even though the rest of the constraint is clearly inconsistent. For example, take the agent `ask(p(X,1)=p(2,2)) ->` A when X is unbound. The second argument of p obviously contradicts the equality constraint, but this agent will nevertheless suspend waiting for the value of X. The semantics of cc suspends this ask infinitely, so the behavior is correct but we may take more space than necessary. The advantage of this scheme is that it is easy to program. The blocked asker implicitly remembers what sub-problems remain to be solved in the constraint. When it is resumed because its waiting condition is satisfied, it simply continues where it left off.

The second alternative tries to find possible inconsistencies as soon as possible. At the end of the consistency check, there remains a list of sub-problems that were not solved because they would have blocked. There is a choice for which variable to suspend on. One can choose the first variable in the first sub-problem, but other variables of interest may become instantiated before that one which may render the constraint inconsistent. This cannot be recognized immediately since the process is waiting for the instantiation of the first variable. In that case, it is hard to justify the effort spent in the first phase when checking the consistency. One can disjunctively wait on all the variables in the unsolved sub-problems and resume as soon as on of those is instantiated. Then one must reconsider the sub-problems in which the newly instantiated variable appear.

One has to balance the advantage of aborting an agent early through the detection of some inconsistency and the work performed to find the inconsistency. For that reason, Taskell's implementation follows the first scheme.

Naive arithmetic. The operators in naive arithmetic wait until their arguments are instantiated before they compute the result. This is done by implied asks on the arguments. There are many alternatives for how to handle the implied asks and returning the result.

1. The naive arithmetic operators like + are data constructors. Asks and tells must handle the implied asks themselves. This complicates solving asks and tells unnecessarily specially for tells because they must be ready to suspend if an implied ask suspends, whereas they could not suspend before.

2. The naive arithmetic operators are constructors and the implied asks are compiled in place into explicit asks. The compiler must know the implied asks which make them harder to change. This has the other disadvantage that it increases the program text.

3. The naive arithmetic operators are functions that perform the implied asks and return the result of a constructor to describe the constraint. The asks and tells must handle the constraint themselves but they are easier to solve since the implied asks are guaranteed to hold. This is how **Taskell** handles relational operators like <.

4. The naive arithmetic operators are functions that perform the implied asks and return the arithmetic result. The treatment of arithmetic is uniform for both asks and tells because they never see the arithmetic operators, they only see the integer results. This assumes the arguments to functions are evaluated before the function is applied, which is a reasonable assumption. **Taskell** uses this last alternative to implement naive arithmetic.

Compound constraints. **Taskell** has conjunction in both ask and tell and disjunction in ask. These are easy to implement because they can be rewritten simply in terms of other agents. The agent `ask(`c_1 `and` c_2`) ->` A can be implemented with `ask(`c_1`) -> ask(`c_2`) ->` A. The agent `tell(`c_1 `and` c_2`)` can be implemented with `tell(`c_1`) || tell(`c_2`)`. Disjunction in ask is similar to a choice with the same agent in all the branches, i.e., the agent `ask(`c_1 `or` c_2`) ->` A can be implemented with

choice { **ask**(c_1) => A + **ask**(c_2) => A}. Using the sequentiality of the target language, this simplifies to **choice** { **ask**(c_1) => () + **ask**(c_2) => ()}; A where () means do nothing. Taskell does not have disjunction in tell constraints because it would need disjunction in its constraint system. Disjunction is tells cannot be easily rewritten in terms of other agents.

6.2 Overview of the Implementation

The implementation of Taskell went through some changes. The study of QDJanus was beneficial but it is sequential and we wanted a parallel implementation. At first, we decided to solve asks in parallel and tells sequentially. There was a process called the teller. Agents sent their tell requests to the teller to be solved. The teller was a sequential process so there was only one tell performed at a time. A variable was an ML reference because its value needed to be changed at the time it was instantiated. The value could also be changed without instantiation because an unbound variable contained a list of agents waiting foi its instantiated value. CML could guarantee atomic assignment in the following sense. If an asker reads a variable while a teller is assigning a new value, the asker would read the old or the new value but not a mix of the two. The teller also handles requests to suspend until a specified variable is instantiated. The teller would have to resume these agents because it is the only process that can instantiate a variable.

Unfortunately, references are not guaranteed to work in the parallel implementation of CML, the atomicity of assignment is simply a side-effect of the current sequential implementation. The atomicity could be guaranteed with the use of locks on variables. Unfortunately, locks are not built into CML, they must be simulated with processes. This means, you send an acquire request to the lock, do the operation on the variable, and then send a release request to the lock. Since a variable needs a process, we could put the state of the variable in the process and let it respond directly to requests for variable operations like `get_value` and `set_value`. Since a variable process is sequential, there would only be one request executed at a time,

Compiler modules	Translator Parser Scanner
Run-time system	Translator Variable Domain
User program	Prog
Utilities	Utility

Figure 6.1: Separation into modules

giving the same effect as a lock. The new scheme gives us the apparatus we need to solve asks and tells in parallel, we can expect it will work on distributed memory parallel machines because it does not rely on unsafe properties of references.

The **Taskell** compiler reads a **Taskell** program from a file and produces a Concurrent ML program in another file. The program can be run by loading it into **CML** with the **Taskell** run-time system already loaded. The compiler is made up of a scanner, a parser and a code generator called the translator. The run-time system is made up of the solver and a module implementing variables as threads. Figure 6.1 lists the modules in the implementation. The Domain module describes the types of values and the constraints that form the constraint system. The Utility module contains useful functions like `member` that are not in the ML library. Figure 6.2 describes the dependencies between the various modules. From the figure, we see the compiler and the run-time system are somewhat independent.

6.3 Concurrent ML

Concurrent ML was designed recently by John Reppy at Cornell University[Rep90, Rep91]. It is a superset of Standard ML of New Jersey that supports concurrency and communication between processes. The current version timeshares a single UNIX

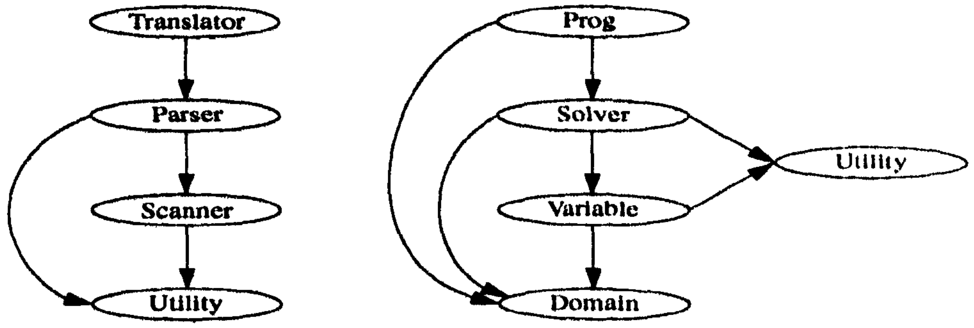

Figure 6.2: Dependencies between modules

process for all Concurrent ML threads. A truly parallel version running on a multi-processor is under way When it becomes available, we automatically have a parallel implementation of Taskell running on a parallel machine.

Standard ML is a higher-order polymorphically typed language originating from Edinburgh[MtH90, Mt91]. Type inference makes type declarations largely unnecessary while remaining strongly typed. This saves a lot of time in programming but may make error messages hard to understand. ML consists of a functional and an imperative subset. Most variables are given a value through binding, the value of these variables cannot be changed after it is bound. Assignment is allowed on variables but they must be declared of type `ref`. Variables of type `ref` are called references.

Concurrent ML is built around the model of communicating sequential processes and message passing. Communication is synchronous, i.e, a send or a receive blocks until its partner arrives. **CML** processes are created dynamically. Messages are sent on channels that are also created dynamically. It is possible to send channels in messages, making the communication network highly dynamic. Furthermore, the communication operations are first-class values. It is possible to build descriptions of protocols and execute them later, and even send them around. This type of value is called an event. Executing an event is called synchronizing on an event and is done with the function `sync (`*event*`)`. One advantage of events is the added expressive power of the selective communication in Concurrent ML for which the set of choices can be computed dynamically. This is in contrast to Ada in which the set of choices in a select are frozen at compile time because it appears textually in the program. Occam's ALT statement shares the same limitation. Communication as a first-class value is the topic of John Reppy's Ph.D. thesis[Rep92].

CML reclaims processes as part of its garbage collection. A process is reclaimed if it is blocked on a channel, and no other active process has access to that channel. These processes would block forever because there can never be a process available to form the connection. Process reclamation simplifies programming because the programmer is relieved from terminating those processes explicitly. Without reclamation, terminating processes would be difficult because **CML** does not have an abort statement: a process can terminate itself but it cannot abort other processes.

CML processes are implemented with continuations. Process creation and task switching are particularly efficient in **CML**, contrary to other continuation based concurrency schemes. The memory organization of **CML** puts everything on the heap and this makes a call with current continuation a constant time operation.

The module system of Standard ML is based on the concepts of structures, signatures and functors. A modules is a structure, the interface of a module is a signature. A signature is the type of a structure because it gives the type of the components of the module. A functor is a parameterized module, it can be applied to arguments to yield a structure. These names were chosen by analogy to their mathematical meaning. For example, a functor is a form of higher-order function, hence its name. Functors are compiled once, but can be applied any number of times. To implement separate compilation, you write all your modules as functors and you apply them to link your application together.

We chose Concurrent ML for the implementation of Taskell because it is a very high level functional language with novel communication facilities. We have greatly simplified the implementation by choosing Concurrent ML as the target language of our translator. Many features of Taskell are inherited from similar features of **CML**. In particular, Taskell's lexical scoping, procedure calls, garbage collection, process management, synchronization and termination detection all come from **CML**. Other features of **CML** not shared by **Taskell** were also useful, including communication as first-class values and interprocess communication.

6.4 The Taskell Compiler[1]

The **Taskell** compiler is made up of a scanner, a parser and a code generator called the translator. The parser requests the tokens from the scanner one at a time. The parser uses recursive descent to build an abstract syntax tree. The complete tree is sent to the translator, which uses a tree walk to translate it to Concurrent ML in one pass. The parser tries to recover from errors by looking ahead for the expected token or some synchronizing token. The error recovery is based on the exception mechanism of Standard ML. The scanner and parser can print debugging messages if a flag is a turned on.

The compiler maps **Taskell** features to **CML** features. Figure 6.3 gives the correspondence between them. Most items are self explanatory except for an agent continuation: this is where a blocking ask will resume if its constraint is entailed. For example, the agent continuation of `ask(X=1) ->` A is the agent A. An agent is a process evaluating an expression, while an agent continuation is an expression that may be evaluated by an agent. **CML** has continuations, but they are not necessary in this case because λ-abstraction is sufficient.

Figures 6.4 to 6.6 show in detail the translation schemas applied by the translator. T is the translation function from **Taskell** to **CML**. The compiler has equivalent schemas from the abstract syntax tree to **CML**. The function T on the left hand side expands to the right hand side. The function T on the right hand side stands for the result of applying T to the argument. Characters in `typewriter` font appear explicitly. A word in *italics* stands for some construct. Words in *italics* on the right hand side stand for some string taken from the argument of T

All elements of the domain have type `value` once in **CML**. The `value` data type is shown in figure 6.7. The constructor Tree has the type `value`. Integers are

[1]The scanner and parser were inspired by the SIL compiler written by Prof. Laurie Hendren.

Taskell	CML
program	structure
procedure definition	function definition
agent	thread
agent continuation	λ-expression
new	`let` newvars `in` ...`end`
ask	function call to `ask`
tell	function call to `tell`
choice	function call to `choice`
procedure call	function call
parallel combinator	series of spawns
constraint	data constructors
naive arithmetic	function calls

Figure 6.3: Correspondence between Taskell features and **CML** features

wrapped in the constructor `Int` to give them the type `value`. Variables are also of type `value` because they can be used where constants can be used. The constructor for variables is `Var`. It is not visible in figures 6.4–6.6 because it is hidden in the function `new`. The constructor `FN` in the schema for `||` and the constructor `Names` in the schema for `main` are needed to correctly infer the types of their argument. The `request` data type is explained fully in section 6.6.

The translator outputs a structure called `Prog` containing two functions. The function `main` is the translation of the Taskell program. The function `run` is needed to start execution of `main`. A **CML** program is started by calling the function `doit` with two arguments, the first argument is a function to be evaluated by the initial process, it takes no arguments and returns no value. The second argument is the time slice for preemptive scheduling. **CML** requires a function like `doit` because it is not yet possible to change the read-eval-print loop in Standard ML. The Taskell program is started by calling `Prog.run ()`. The appendix shows the output of the compiler

```
T[proc main($v_1$, ..., $v_n$) $procs$ begin $A$ end] ≡
            structure Prog =
              struct
                local open Domain Solver in
                  fun main () = let
                      val $v_1$ = new()
                       ⋮
                      val $v_n$ = new()
                      T[$procs$]
                  in
                      answer (Names [($v_1$, "$v_1$"), ..., ($v_n$, "$v_n$")]);
                      T[$A$]
                  end
                  fun run _ =
                      RunCML.doit(main, SOME 20)
                end
              end
```

```
T[$proc_1$, ..., $proc_n$] ≡ fun T[$proc_1$]
                          and T[$proc_2$]
                           ⋮
                          and T[$proc_n$]
```

```
T[proc $name$($v_1$, ..., $v_n$) $procs$ begin $A$ end] ≡
            $name$($v_1$, ..., $v_n$) = let
              T[$procs$]
            in
              T[$A$]
            end
```

Figure 6.4: Translation schemas for procedures.

```
T[ask(c) -> A] ≡ ( ask(T[c]);
                    T[A]
                  )


T[tell(c)] ≡ tell(T[c])


T[new v₁, ..., vₙ in A] ≡ let
                              val v₁ = new ()
                               ⋮
                              val vₙ = new ()
                           in
                              T[A]
                           end


T[{A₁ || ··· || Aₙ}] ≡ ( fork (FN (fn () => T[A₁]);
                           ⋮
                          fork (FN (fn () => T[Aₙ₋₁]);
                          T[Aₙ])
                        )


T[choice {ask(c₁) => A₁ + ··· + ask(cₙ) => Aₙ}] ≡
             choice [
                     (fn (ev) => (guarded(T[c₁], ev); T[A₁])),
                       ⋮
                     (fn (ev) => (guarded(T[cₙ], ev); T[Aₙ]))
                    ]


T[name(a₁, ..., aₙ)] ≡ name(T[a₁], ..., T[aₙ])
```

Figure 6.5: Translation schemas for agent combinators.

T⟦c_1 `and` c_2⟧ ≡ `(` T⟦c_1⟧ `And` T⟦c_2⟧ `)`

T⟦c_1 `or` c_2⟧ ≡ `(` T⟦c_1⟧ `Or` T⟦c_2⟧ `)`

T⟦`isint(`e`)`⟧ ≡ `Isint(`T⟦e⟧`)`

T⟦`istree(`r`,` n`,` e`)`⟧ ≡ `Istree("`r`",` n`,` T⟦e⟧`)`

T⟦$e_1 = e_2$⟧ ≡ `(` T⟦e_1⟧ `==` T⟦e_2⟧ `)`

T⟦e_1 *op* e_2⟧ ≡ `(` T⟦e_1⟧ *opop* T⟦e_2⟧ `)`

T⟦*integer_constant*⟧ ≡ `Int(`*integer_constant*`)`

T⟦*name*`(`a_1`, ...,` a_n`)`⟧ ≡ `Tree("`*name*`",` n`, [` T⟦a_1⟧`, ...,` T⟦a_n⟧ `])`

T⟦*name*⟧ ≡ *name*

Figure 6.6: Translation schemas for constraints and values.

```
datatype value = Unbound
               | Bool of bool
               | Char of string
               | Int of int
               | Tree of string * int * value list
               | Var of variable

and   variable = Variable of request CML.chan
```

Figure 6.7: The ML datatype for Taskell values

on two different programs as well as a trace of their execution.

6.5 The Run-time System of Taskell

The translator outputs calls to the functions: `ask`, `tell`, `new`, `choice`, `fork` and `answer`. These are defined in the solver and imported by the statement `open Solver`. The program produced by the compiler is not functional, as can be expected since it closely resembles the source program. The following paragraphs describe the CML functions that are used in the translation schemas.

The function `ask` takes a constraint c and returns only when c is entailed by the store. `Ask` solves the constraint itself by sending messages to variables. It may abort the agent if it discovers that the negation of c is entailed by the store. It seems that ask takes a function and an agent continuation as argument `ask(`c`, `A`)`, but it is simpler to use the sequentiality of the target language and implement this with `(ask(`c`); `A`)` instead. This trick is not applicable to guards.

The function `tell` takes a constraint c and augments the store with this new constraint. `Tell` solves the constraint itself by sending messages to variables. As it does this, it checks the consistency of c in the store. If the store becomes inconsistent, it aborts the whole program with an error.

The function `new` takes no arguments. It creates a new variable and returns it wrapped in the constructor `Var` to give it the type `value`. Part of creating a variable involves launching a process to handle the requests for that variable.

The function `choice` takes a list of branches as arguments. A branch is a function that solves a guard and possibly executes its guarded agent. `Choice` forks one guard process for each branch. The guards report back to the choice agent which picks the successful branch. That branch is then allowed to execute its guarded agent. This leaves the other guards pending, they will be garbage collected by CML eventually. There are no provisions for tell guards internally.

The argument of a branch is a Concurrent ML event, it describes the communication between the guard and the choice agent. The event is built by the choice agent and sent to the branch when it is forked. A branch uses its event to report success to the choice agent. If the communication succeeds, it means this is the chosen branch and it can execute its guarded agent. It is very important that a branch be a function because it delays the evaluation of the implied asks in the guard until the branch is forked, otherwise an implied ask might suspend the whole choice agent before it has time to launch the branches.

The choice combinator is difficult to implement inside only one process. If all the guards suspend, the process will need disjunctive wait to wake up whenever a guard has more information. It is easier if each guard is executed by a new process. The agent doing the choice remains as a manager for the guard solvers. The guards report to the choice manager. The manager determines the outcome by picking an open guard and executing its guarded agent. The other guard processes either abort themselves or they are reclaimed when they try to report to the manager.

The choice combinator in Taskell is implemented in two different ways. The first implementation picks the first guard to terminate successfully. This is the behavior of committed-choice concurrent logic programming languages. Solving a guard is simplified because it does not have to report failure or suspension. A guard can be implemented directly with a call to `ask`, followed by a message send to the manager to report that this guard is open. The `ask` will return only if the constraint is entailed, so the report will be sent only if the **ask** is successful. The first guard to report is the one picked by the manager.

The second implementation waits until *all* guards are either blocked or terminated, then it picks one at random among those who terminated successfully. If none terminated successfully and there are blocked guards then it blocks, otherwise it aborts. This implementation tries to be as fair as possible. Unfortunately, it cannot use **ask** directly because it needs to report success, suspension or failure to the manager. The code to solve a guard is almost identical to solving an ask except for these small differences.

The function **fork** takes an agent continuation and forks a new process to evaluate it. It returns immediately to the caller. This is a call to the **CML** function **spawn**, but it returns nothing instead of the new process ID.

The function **answer** takes a list of pairs as argument. Each pair is a variable with its printable name. **Answer** is the mechanism by which the system knows which variables form the result at the end of the execution. The printable names are needed because they are not available at run-time.

The run-time system must detect the termination of the program to print the result of the computation at that time. **CML** has a built-in termination detection algorithm for robust termination of programs. The function `logServer("serverName", initFn, termFn)` informs **CML** that a server should be initialized when the program starts and finalized at termination. The function `initFN` will be called to initialize the server at the beginning of the execution. The function `termFN` will be called to finalize the server at termination. Taskell uses this facility to print the result of the computation. The termination function of Taskell receives the list of variables to print from the function **answer** in **main**. The values of the variables are printed one to a line in the format `Name = Value`. Taskell tries to eliminate all variables from the right-hand-side by printing their value, but it cannot eliminate unbound variables. It invents new names for these in the following sequence `T1`, `T2`, and so on. A new name is generated only if the equivalence class doesn't have a name already. The number of equation printed is always the same as the number of variables in the result. Taskell does not try to minimize the output by adding more equations.

6.6 Implementation of Variables

Section 6.2 explained why variables were implemented as processes. The value of a variable changes from `unbound` to some other value when it is instantiated. In Standard ML one would use a reference and assign the value to the variable when it is instantiated. In Concurrent ML, shared references are not guaranteed to work in the presence of concurrency. Since we need atomicity in some variable operations and the only form of synchronization in Concurrent ML is through communication, we implement variables as processes. Agents must send messages to variables to request an operation on variables. This section develops the protocol used between agents and variables. There are many details one has to be careful about in order to assure atomicity and to avoid deadlock.

For an agent, a variable `X` is represented by a channel. The agent can get the value of the variable by sending a request on the channel. The request can be `current_value` or `instantiated_value`. The variable `X` is implemented with a process reading its channel for requests sent by agents. The variable responds to a `current_value` request by returning the value of the variable at this time regardless if it is instantiated or not. Non-instantiated variables return the value `Unbound`. The response to `instantiated_value` requests depends on the status of the variable. An instantiated variable returns its value immediately. An unbound variable puts the request in a queue. The requester becomes an instantiation suspending agents of `X` Other types of requests may make the agent an equation suspending agent of `X`.

The process implementing a variable is actually responsible for the whole equivalence class. When a variable `X` is equated to the variable `Y` and they are both unbound, then the equivalence class of `X` is merged into the equivalence class of `Y` thereby delegating the responsibility of `X` to `Y`. The process for `X` disappears. This kind of delegation is trivial to implement in **CML** being one of the main applications of communication as a first-class value.

The communication behavior of a variable is represented as a value called an event. Each variable process carries its event as part of its state. The event of `X` describes the communication behavior of the variable `X`, similarly, the event of `Y` describes the communication behavior of the variable `Y`. The function `sync` applied to an event executes the behavior described by that event. Synchronizing on the event of `X` returns a request for the variable `X`.

The equivalence class of `X` can be merged with the class of `Y` by building an event describing the behavior resulting from behaving as `X` and `Y`. In particular, the new class must accept requests from variables in the class of `X` *and* requests from variables in the class of `Y`, but requests are read one at a time, so the next one will come from either the class of `X` *or* the class of `Y`. Therefore the new class must behave as the event of `X` or the event of `Y` depending on the availability of requests. The function `choose` in **CML** produces an event describing exactly this behavior. The new compound event we assign to `Y` is thus `choose [ eventX, eventY ]`. The variable `Y` will now accept requests from all channels it was already handling plus those previously handled by `X`. The variable `Y` does not have to know which channels were handled by `X` because this is all encapsulated in `eventX`. At first, a variable will only listen to one channel but other channels are added every time a class is merged in The reason delegation is so easy is because `choose` computes its branches dynamically whereas for example `select` in Ada has a fixed set of branches determined from the program text at compile time.

The protocol between agents and variables also involves messages between two variables. To merge the equivalence class of `X` into `Y`, `X` must send its event to `Y`. The variable `X` must also send its suspending agents. The instantiation suspending agents of `X` are added to the instantiation suspending agents of `Y`. The equation suspending agents of `X` may or may not be released depending if the variable they are waiting for is in the class of `Y`, similarly, the equation suspending agents of `Y` may or may not be released depending if the variable they are waiting for is in the class of `X`. This can only be determined if we keep a list of the variables that are members of the class. The representation of variables in this list is the same as the one used in agents. Each item in the list is a channel. Thus variable processes have a list of channels in their state describing the variables in its equivalence class.

The agent `ask(X=Y) -> A` needs to test the equality of the two variables. Two variables are equal if they have the same value, or if they are in the same class. The agent may assume that `X` and `Y` are instantiated, and if not, it can check the equivalence classes:

> `valueX` = `fetch_value(X)`, `X` is unbound
>
> `valueY` = `fetch_value(Y)`, `Y` is unbound
>
> `X` and `Y` are instantiated to 2 by another agent
>
> `classX` = `fetch_class(X)`
>
> `classY` = `fetch_class(Y)`
>
> compare `classX` and `classY`

The classes may be different but the variables are equal because their value is 2. This problem arises because fetching the value of the variable and the fetching the class are not atomic operations. The solution is to use a function that returns both the value and the class at the same time:

> `(valueX, classX)` = `current_value(X)`,

This means that `current_value` returns two values, the first is bound to `valueX` and the second is bound to `classX`.

Testing the equality of two unbound variables brings up the problem of representing an equivalence class and comparing two classes to see if they are the same. The natural choice for an equivalence class is to represent it by the list of its variables. The class of `X` is obviously equal to the class of `Y` if the two list of variables are the same. Nevertheless the lists could be different and yet `X` and `Y` could be in the same class. The problem arises because fetching the equivalence class of two variables is not an atomic operation.

Consider the following scenario: the agent A_1 executes `ask(X=Y) -> A`, the agent A_2 executes {`tell(X=Z) || tell(Z=Y)`} and the three variables `X`, `Y` and `Z` are all unbound. The following actions happen in this order:

A_1: let `(valueX, classX) = current_value(X)`, `X` is unbound

A_2: merge class of `X` into `Z`

A_2: merge class of `Z` into `Y`

A_1: let `(valueY, classY) = current_value(Y)`

Now the variable `X` is in the class of `Y` but `classX` $\neq$ `classY`. A better test would be to check membership of `X` in the list `classY` and if it is not, then A_1 becomes an equation suspending agent of `Y`. We have to fetch the class of `X` in any case because we don't know if it is uninstantiated. We can optimize the test by evaluating `classX = classY` or else `member(X, classY)`. Note that `member(Y, classX)` would not work here because `classX` has outdated information.

Consider the simplified agents A_1 executing `ask(X=Y) -> ` A and A_2 executing `tell(X=Y)`. Both variables are unbound and the following actions happen in that order:

A_1: let `classX = fetch_class(X)`

A_1: let `classY = fetch_class(Y)`

A_1: send a `Wait_merge(X)` request to `Y`

A_2: merge class of `X` into `Y`

A_1: `Y` receives the `Wait_merge(X)` request

The variable `Y` receives a request to wait until `X` is in this class but `X` is already in the class. The variable `Y` must check that `X` is not in its list of variables before queuing the new suspending agent.

This same precaution is necessary with merge requests. Consider the agent A_1 executing `tell(X=Y)` and the agent A_2 executing the same thing. These agents must fetch the class of the variables to see if they are already instantiated, but in this case both variables are unbound. Let the following actions happen in order:

A_1: let `(valueX1, classX1) = current_value(X)`

A_2: let `(valueX2, classX2) = current_value(X)`

A_1: let (valueY1, classY1) = current_value(Y)

A_2: let (valueY2, classY2) = current_value(Y)

A_1: send a Merge(X) request to Y

A_1: Y merges X into its class

A_2: send a Merge(X) request to Y

The variable Y will receive a Merge(X) request from A_2 but X is already in its class. We do not want the list of variables to contain duplicates nor do we want to have more complicated events than necessary. More importantly, we do not want to release the suspending agents of X twice. For that reason, when a variable receives a merge request, it should check first if the variable is not already in the list of variables.

Deadlock can occur if two variables try to mutually merge themselves. Consider the agent A_1 executing tell(X=Y) and the agent A_2 executing tell(Y=X). Both variables are unbound. Let the following actions happen in order:

A_1: let (valueX1, classX1) = current_value(X)

A_2: let (valueY2, classY2) = current_value(Y)

A_1: let (valueY1, classY1) = current_value(Y)

A_2: let (valueX2, classX2) = current_value(X)

A_1: send a Merge(X) request to Y

A_2: send a Merge(Y) request to X

A_1: Y sends a Merge_me(Y) request to X

A_2: X sends a Merge_me(X) request to Y

This causes deadlock because Y expects X to take over, while X expects Y to take over. The deadlock can be avoided by imposing a partial order on variables. The agent tell(X=Y) may go ahead if X $\geq$ Y, otherwise it is rewritten as tell(Y=X). Alternatively, we can impose an ordering on equivalence classes. Unfortunately, CML has no notion of ordering among channels. Variables cannot be compared, nor can we compare equivalence classes.

CML does have a notion of ordering among process ID's. We redefine the representation of the equivalence class as a pair formed by a process ID and a list of variables. That process ID comes from the process that answered the request for the `current_value`. The ordering of equivalence classes is based exclusively on the process ID field disregarding the list of variables. We can also redefine the equality test for equivalence classes. Two classes are the same if the process ID's are the same or if the list of variables intersect. The process ID is the same if the same process is responsible for these two variables. The list of variables may be different, but since variables are always added and never removed from the list, this means they are in the same class. When merging `X` into `Y`, the process for `X` will disappear, so the class of `X` will have a different process ID before and after the merge. We can still recognize efficiently that these two classes are the same by putting the variables of `X` at the beginning of the list of variables in `Y` (remember that the process ID of `Y` is the one preserved). Therefore the two lists will intersect on the first variable. A full check of intersection is expensive. We can have an efficient conservative estimate for when classes are equal by looking at the process ID's and the first variable in the list. It is conservative because sometimes the classes will intersect but it will not be recognized.

So far, the variables were always unbound. We will see that `Wait_merge` and `Merge` requests may fail because the variable has been instantiated by the time the request was received. When this happens, an agent must fall back to the case when the variable is instantiated in the first place. Consider the agent A_1 executing `ask(X=Y) ->` A and the agent A_2 executing `tell(Y=3)`. Both variables are unbound at the beginning. Let the following actions happen in order:

A_1: let `classX = fetch_class(X)`

A_1: let `classY = fetch_class(Y)`

A_1: send a `Wait_merge(X)` request to `Y`

A_2: instantiates `X` to 3

A_1: `Y` receives a `Wait_merge` request

The variable `Y` receives the `Wait_merge` request after it was instantiated. The constraint `X=Y` may be entailed if `X` is instantiated to 3, it does not have to be in the equivalence class of `Y`. The `Wait_merge` request will fail, this will give a chance to the agent A_1 to become an instantiation suspending agent for `X` Similarly, instantiating a variable may fail. For example, the agent `tell(X=2)` may try to instantiate `X` but by the time the request arrives to `X`, it is already instantiated. The agent should impose the equality of the two values.

We have seen we can merge the classes of two unbound variables and we can instantiate a variable with a value. We can also instantiate an unbound variable by merging it with an instantiated variable. We first send a `Merge_into(Y)` request to the unbound variable `X`. This will fail if `X` was instantiated by the time this request was received, in that case, the agent should impose the equality of the two values. Otherwise, `X` sends a `Merge_me(X)` request to the instantiated variable `Y`. This will always succeed. All suspending agents of `X` are released by `Y` because the variable is now instantiated. The event of `X` is merged with the event of `Y`, and `X` disappears.

A further improvement would be to merge two instantiated variables when their values are the same. This can be confirmed by an ask or be imposed by a tell. The current implementation does not attempt to do this because it never changes the value of a variable once it is instantiated.

We summarize the interface between agents and variables. When we say a function fails we mean it returns an indication that the operation was not performed

The function `current_value(X)` returns the value and the class of the variable `X`. It never suspends because it may return the value `Unbound`. The function `instantiated_value(X)` returns the instantiated value of the variable `X`. It will suspend until `X` is instantiated.

The function `set_value(X,V)` instantiates the variable `X` to the value `V`. If `X` is already instantiated by the time this request arrives, then `set_value` fails and the user should check the consistency of the value of `X` and `V`.

The function `merged_or_instantiated(X,Y)` suspends until `X` and `Y` are equated, i.e., until they are merged into the same equivalence class. This routine could fail when X is instantiated, but instead it returns the value of X. This routine is used when an agent is asking if X=Y and X and Y are both unbound and not in the same equivalence class. The agent becomes an equation suspending agent of X for Y. This routine returns if X is instantiated because X and Y may have equal values but not be in the same class.

The function `merge_class(X,Y)` merges the equivalence class of X and Y. The user should guarantee `classX < classY` if both variables are unbound. This routine sends the request `Merge_into(Y)` to X. If X is unbound when it receives this request, then it sends the request `Merge_me(X)` to Y. The equivalence class of X will be merged into the class of Y. This always succeeds because you can always merge an unbound variable with another variable and X cannot become instantiated at that stage (it is busy with the merge request). Figure 6.8 shows the algorithm in detail If X is instantiated by the time it receives the `Merge_into` request then `merge_class` fails. The user can try to `merge_class(Y,X)` to instantiate the variable Y by merging it into the class of X. This will fail if Y was instantiated by the time this request arrived to Y. Then the user should check the consistency of the value of X and Y. This algorithm is given with the rest of the unification algorithm in figure 6.9 to figure 6.11.

6.7 Debugging

The implementation has a set of switches to turn debugging statements on and off. There are separate switches for the scanner, parser, solver and variable modules. Of particular interest are the messages from the solver and the variables because they give a trace of the execution of the Taskell program. The solver has messages for the operators ask, tell and choice. A message is always preceded by the process ID of the process writing the message. An ask will print two messages, one when it is called, and one for the result. These two messages will be separated by messages from other agents, but they can be paired by looking at the process ID. The result can be `entailed` or `inconsistent`. A tell will also print two messages, one when it is called and one for the result. The result can be `accepted` or `inconsistent`. An

This code is executed by the variable X:

```
When receive a Merge_into(Y) request
    if member(Y, ListOfVarsX) then
            X is already merged with Y so nothing needs to be done
    else if X is instantiated then
            reply that X is already instantiated
    else
            X sends a merge_me(X) request to Y with its event, instan
            tiation and equation suspending agents and listOfVarsX.
            This always succeeds because X cannot be instantiated at
            that stage, it is busy executing this line
```

This code is executed by the variable Y:

```
When receive a Merge_me(X) request
    eventY = choose [eventX, eventY]
    if Y is instantiated then
            release X's instantiation suspending agents and
            release X's equation suspending agents
    else
            append X's instantiation suspending agents to Y's instantia
            tion suspending agents
            let Susp be the empty list
            for each equation suspending agent A of X
                    if member(A, ListOfVarsY) then
                            release A
                    else insert A into Susp
            for each equation suspending agent A of Y
                    if member(A, ListOfVarsX) then
                            release A
                    else insert A into Susp
            let Y's equation suspending agents = Susp
            let ListOfVarsY = append(ListOfVarsX, ListOfVarsY)
```

Figure 6.8: The algorithm for merge_class(X,Y)

```
case int($i$) = int($j$)
    check $i = j$
case int($i$) = var($X$)
    ask var($X$) = int($i$)
case int($i$) = anything else
    inconsistent

case tree($r1$,$n1$,$a1$) = tree($r2$,$n2$,$a2$)
    check $r1 = r2$ and $n1 = n2$ and ask $a1_i = a2_i \quad \forall i \ (0 \leq i \leq n1)$
case tree($r1$,$n1$,$a1$) = var($X$)
    ask var($X$) = tree($r1$,$n1$,$a1$)
case tree($r1$,$n1$,$a1$) = anything else
    inconsistent

case var($X$) = var($Y$)
    let ($valueX$, $classX$) = current_value($X$)
    let ($valueY$, $classY$) = current_value($Y$)
    if $valueX \neq$ unbound and $valueY \neq$ unbound then
        ask $valueX = valueY$
    else if $valueX =$ unbound and $valueY \neq$ unbound then
        ask instantiated_value($X$) = $valueY$
    else if $valueX \neq$ unbound and $valueY =$ unbound then
        ask $valueX$ = instantiated_value($Y$)
    else
        if merge_or_instantiated($X$,$Y$) = instantiated then
            if merge_or_instantiated($Y$,$X$) = instantiated then
                    ask instantiated_value($X$) = instantiated_value($Y$)
            else done
        else done

case var($X$) = anExpr
    ask instantiated_value($X$) = anExpr
```

Figure 6.9: The unification algorithm for asks.

```
case int($i$) = int($j$)
    check $i = j$
case int($i$) = var($X$)
    tell var($X$) = int($i$)
case int($i$) = anything else
    inconsistent store

case tree($r1$,$n1$,$a1$) = tree($r2$,$n2$,$a2$)
    check $r1 = r2$ and $n1 = n2$ and tell $a1_i = a2_i \quad \forall i\ (0 \leq i \leq n1)$
case tree($r1$,$n1$,$a1$) = var($X$)
    tell var($X$) = tree($r1$,$n1$,$a1$)
case tree($r1$,$n1$,$a1$) = anything else
    inconsistent store

case var($X$) = var($Y$)
    let ($valueX$, $classX$) = current_value($X$)
    let ($valueY$, $classY$) = current_value($Y$)
    if $valueX \neq$ unbound and $valueY \neq$ unbound then
        tell $valueX = valueY$
    else if $valueX =$ unbound and $valueY \neq$ unbound then
        if merge_class($X$,$Y$) = instantiated then
                tell instantiated_value($X$) = $valueY$
        else done
    else if $valueX \neq$ unbound and $valueY =$ unbound then
        if merge_class($Y$,$X$) = instantiated then
                tell $valueX$ = instantiated_value($Y$)
        else done
    else
        swap $X$ and $Y$ if $X < Y$
        if merge_class($X$,$Y$) = instantiated then
                if merge_class($Y$,$X$) = instantiated then
                        tell instantiated_value($X$) = instantiated_value($Y$)
                else done
        else done
```

Figure 6.10: The unification algorithm for tells.

```
case var($X$) = anExpr
    let (*valueX*, *classX*) = current_value($X$)
    if *valueX* $\neq$ unbound then
        tell *valueX* = anExpr
    else if set_value($X$,anExpr) = instantiated then
        tell *valueX* = anExpr
    else done
```

Figure 6.11: The unification algorithm for tells (continued).

inconsistent tell will abort the execution. A choice will print one or two messages. The first one gives the number of guards and the second one gives which guard is selected when there are two or more guards to choose from. Each guard will also print two messages like an ask. Figure 6.12 summarizes the messages from the solver.

There is a switch to turn messages from all variables on or off. When it is on, variables will print the type of request they receive as they receive them. The requests can be `current_value`, `instantiated_value`, `set_value`, `wait_merge`, `merge_into` or `merge_me`. The process ID of the requester is printed on the left, the process ID of the variable is printed next, followed by the type of request. The arguments of the requests are not printed because it would produce too much output. Figure 6 13 summarizes the messages from variables.

Printing the trace messages in the solver requires the value of the variables in the constraints. In turn this would generate trace messages from variables that are not applicable to any agent combinator. Instead, the values of variables in trace messages are read with the request `untraced_value` that behaves as `current_value` but does not print a trace message.

Two trace messages will never be intertwined because a trace message is built in a string and printed with a single print statement and CML input/output statements are atomic.

```
[process-ID]: ask(constraint)
[process-ID]: ask(constraint):  entailed
[process-ID]: ask(constraint):  inconsistent

[process-ID]: tell(constraint)
[process-ID]: tell(constraint):  accepted
[process-ID]: tell(constraint):  inconsistent

[process-ID]: choice between n guarded agents
[process-ID]: guard i selected
```

Figure 6.12: Trace messages from the solver

```
[caller-process-ID]: variable [variable-process-ID]: current_value
[caller-process-ID]: variable [variable-process-ID]: instantiated_value
[caller-process-ID]: variable [variable-process-ID]: set_value
[caller-process-ID]: variable [variable-process-ID]: wait_merge
[caller-process-ID]: variable [variable-process-ID]: merge_into
[caller-process-ID]: variable [variable-process-ID]: merge_me
```

Figure 6.13: Trace messages from variables

Chapter 7

Conclusions

This thesis describes a parallel implementation of a concurrent constraint programming language. The constraint system of Taskell is the set of finite trees with equality. This constraint system is very close to first order terms with equality used in most logic programming languages. Taskell turned out to be powerful enough to express a variety of concurrent programming problems, for example, tightly coupled producer-consumer relationships, client-server relationships and non-determinate computations.

As a programming paradigm, the cc framework is a natural outgrowth of logic programming. The ask and tell primitives allow one to capture the notions of synchronization and communication in a perspicuous manner. Thus it is usually clear when one needs to use ask and tell in various situations. Unfortunately, when the constraint system is weak, one is sometimes forced into using convoluted encodings to express simple programming idioms. For example, the agent `ask(∃ X1, X2. X = node(X1,X2)) -> A` where `A` uses `X1` and `X2` must be written in Taskell: `new X1, X2 in {ask(istree(node,2,X)) -> {tell(X=node(X1,X2) || A}}`, because Taskell does not have existential quantifiers in constraints.

Ask accomplishes co-routining between processes and thus at the operational level, one can arrange complex patterns of control flow. However, because of the simple constraint based semantics, the programmer can think in more logical terms, for example viewing parallel imposition of constraints as conjunction and viewing ask as a conditional.

CML is a very effective prototyping language for concurrent problems. Unfortunately, you need a fast machine with a lot of memory for acceptable performance. The paging behavior of **CML** in small memories is particularly bad. Nevertheless we found that **CML** provided most of the concurrency idioms that one needs.

The implementation was simplified enormously by the choice of **CML** as our target language. Many features of Taskell are inherited from **CML**, including lexical scoping, procedure calls, garbage collection, process management, synchronization and termination detection. The coding effort concentrated on the compiler, constraint solving and equivalence classes as processes.

CML allowed us to implement only the features that are specific to Taskell. It is difficult to extract from the code the pieces that are independent of the constraint system because most of these were inherited from **CML**. For that reason, this implementation of Taskell may serve as a model more than a starting point for other cc implementations.

CML more or less forced us to use processes for variables, something we did not want to do originally. Fortunately, processes in **CML** are very lightweight, they take little memory and context switch is fast. Communications as first-class values simplified the merge of two equivalence classes. Delegation of responsibilities is almost trivial in our case. Taskell relies on the implementation of events to realize the binding between variables. There is an extensive body of literature on this problem in concurrent logic programming. Out of the many schemes proposed, we cannot expect **CML**'s will be optimal for this application.

The cc family is parameterized by a choice of constraint system and agent combinators. The agent combinators fix the process language and the constraint system fixes the type of constraints we can work with and the entailment relation. In theory, the process language is independent of the constraint system. This is not, however, bourne out by our experience.

It is tempting to build a cc shell that given a constraint solver and a choice of agent combinators would produce a cc implementation for that instance. We suspect a shell with that much generality is bound to fail because the agent combinators have a big impact on how the constraint solver is written. For example, if the parallel-or combinator is supported by backtracking then the solver should trail the modifications it makes to the store. If tell guards are allowed in a choice, then it needs to attempt a change to the store without leaving a trace if it is not the chosen guard. This can be done with locks on variables together with deadlock avoidance or with copies of the store.

The agent combinators cannot be chosen independently of the constraint solver You cannot add a combinator that was not planned by the solver. The reason a CLP shell is successful is because all instances of CLP share the same set of agent combinators. A cc shell could be useful as long as the solver is tailored to the set of agent combinators. The shell could provide process scheduling and a user interface for example.

Appendix A

Sample Session

This appendix demonstrates the implementation through a sample session. We first explain how to run a Taskell program in our implementation. The user has to load Taskell in Concurrent ML. It is possible to create a version of CML Taskell already loaded but this requires a lot of disk space (4Mb). The program is translated into CML, and the output is left in a file with extension .ml. This file must loaded into CML. The user can choose to turn trace messages on or off by assigning to the debug switches. Finally, the program is launched by evaluating Prog.run ().

In summary, the steps to execute a Taskell program are:

```
% cml
- use "taskell.ml";
- translate "yourProgramName";
- use "yourProgramName.ml";
- Solver.print_trace := true;   (* if you want a trace *)
- Variable.print_trace := true;
- Prog.run ();
```

The sample session is split into two parts. Each part explains a program, lists the source file, shows a trace of execution and then lists the output of the compiler on the source file. The script has been edited to reduce the number of messages when ML is loading files. Chapter 6 explains the meaning of the trace messages, section 6.7

gives a short summary. The **CML** program produced by the compiler is explained in section 6.4.

The first program is a demonstration of trace messages from variables. Showing messages from variables makes the trace more difficult to follow. Normally you would choose to trace only agent combinators, because the trace becomes much more apparent. That is what we do in the second part. The point of this example is to show the messages between variables in a straightforward program. The program is in the file `showvars`, it is translated into Concurrent ML, and the translated output in the file `showvars ml` is loaded The trace messages are turned on for variables and for the solver, and the execution is launched. The program assigns the value `3` to the variable `X` through another variable in its equivalence class. The choice agent suspends until `X` is instantiated. The answer is the value of `X` spelled out: `three`.

The trace can be understood as follows: the variables `Y`, `X1`, `X2`, `X3` and `X` are the processes `[6]`, `[9]`, `[10]`, `[11]` and `[12]` respectively. The agent `tell(X1=X2)` is the process `[14]`, the agent `tell(X3=3)` is the process `[15]`, the agent `tell(X2=X3)` is the process `[16]` and the agent `tell(X=X1)` is the process `[6]`.

At line 4, the variable `X3` gets the value `3`, this is reported on line 8. The agent `tell(X2=X3)` would normally merge `X3` into `X2` because of the ordering on variables, but since it recognizes that `X3` is already instantiated on line 10, it tries the opposite, i.e., it tries to instantiate `X2` by merging it into `X3` on line 13, success is reported on line 17. On line 14, `tell(X=X1)` instructs `X` to merge itself into `X1` because these variables are unbound (line 9 & 12) and `X` is bigger in the variable ordering. Success is reported on line 18. At this stage, `X3` handles `X2` and they are instantiated to `3`, `X1` handles `X` and they are still unbound. `tell(X1=X2)` thinks that `X2` is uninstantiated because it got the value on line 11, before `X2` was instantiated on line 15. Merging `X2` into `X1` will fail on line 19 because `X2` is already instantiated. The agent tries to instantiate `X1` instead by merging it into `X2` on line 20. Success is reported on line 22. This instantiates the variable `X`. The third guard is selected because it is the only consistent one. This instantiates the value of `Y` to `three` and the execution terminates.

```
Script started on Tue Nov 12 08:31:13 1991

% cat showvars

proc main (Y)
begin
  new X1,X2,X3,X in ι
      choice {
          ask(X=1) => tell(Y=one)
        + ask(X=2) => tell(Y=two)
        + ask(X=3) => tell(Y=three)
        + ask(X>3) => tell(Y=big)
      }
   || tell(X1=X2)
   || tell(X3=3)
   || tell(X2=X3)
   || tell(X=X1)
  }
end

% cml
Concurrent ML, version 0.9.5, July 12, 1991
Standard ML of New Jersey, Version 0.71, 23 July 1991

- use "taskel.ml";
[opening taskell.ml]

[opening utility-sig.ml]
[opening scanner-sig.ml]
[opening parser-sig.ml]
[opening translator-sig.ml]
[opening domain-sig.ml]
[opening variable-sig.ml]
[opening solver-sig.ml]

[opening utility.ml]
[opening scanner.ml]
```

```
[opening parser.ml]
[opening translator.ml]
[opening domain.ml]
[opening variable.ml]
[opening solver.ml]

- translate "showvars";

- use "showvars.ml";
[opening showvars.ml]
structure Prog :
  sig
    val main : unit -> unit
    val run : 'a -> unit
  end
[closing showvars.ml]

- Variable.print_trace := true;
- Solver.print_trace := true;




- Prog.run ();

    1  [15]: tell(T1 = 3)
    2  [16]: tell(T1 = T2)
    3  [14]: tell(T1 = T2)
    4  [15]: variable [11]: set_value
    5  [6]:  tell(T1 = T2)
    6  [16]: variable [10]: current_value
    7  [14]: variable [9]:  current_value
    8  [15]: tell(T1 = 3):  accepted
    9  [6]:  variable [12]: current_value
   10  [16]: variable [11]: current_value
   11  [14]: variable [10]: current_value
   12  [6]:  variable [9]:  current_value
   13  [16]: variable [10]: merge_into
```

```
14  [6]:  variable [12]: merge_into
15  [10]: variable [11]: merge_me
16  [12]: variable [9]:  merge_me
17  [16]: tell(T1 = T2): accepted
18  [6]:  tell(T1 = T2): accepted
19  [14]: variable [11]: merge_into
20  [14]: variable [9]:  merge_into
21  [9]:  variable [11]: merge_me
22  [14]: tell(T1 = T2): accepted
23  [13]: choice between 4 guarded agents
24  [20]: variable [11]: instantiated_value
25  [17]: ask(3 = 1)
26  [20]: ask(false = true)
27  [20]: ask(false = true): inconsistent
28  [18]: ask(3 = 2)
29  [19]: ask(3 = 3)
30  [17]: variable [11]: instantiated_value
31  [17]: ask(3 = 1): inconsistent
32  [18]: variable [11]: instantiated_value
33  [18]: ask(3 = 2): inconsistent
34  [19]: variable [11]: instantiated_value
35  [19]: ask(3 = 3): entailed
36  [13]: guard 3 selected
37  [13]: tell(T1 = three)
38  [13]: variable [7]:  set_value
39  [13]: tell(T1 = three): accepted

Answer:

Y = three
Ok
```

```
- ^Z
Stopped

% cat showvars.ml

structure Prog =
  struct
    local open Domain Solver in
      fun main () = let
        val Y = new()
      in
        answer (Names [(Y,"Y")]);
        let
          val X1 = new()
          val X2 = new()
          val X3 = new()
          val X = new()
        in
          (
            fork (FN (fn () =>
                choice [
                  (fn (ev) => (guarded(((X==Int(1))), ev);
                    tell((Y==Tree("one",0,[]))
                  )),
                  (fn (ev) => (guarded(((X==Int(2))), ev);
                    tell((Y==Tree("two",0,[]))
                  )),
                  (fn (ev) => (guarded(((X==Int(3))), ev);
                    tell((Y==Tree("three",0,[]))
                  )),
                  (fn (ev) => (guarded(((X>>Int(3))), ev);
                    tell((Y==Tree("big",0,[]))
                  ))
                ]
            ));
            fork (FN (fn () =>
                tell((X1==X2))
            ));
```

```
                fork (FN (fn () =>
                    tell((X3==Int(3)))
                ));
                fork (FN (fn () =>
                    tell((X2==X3))
                ));
                tell((X==X1))
              )
            end
          end

          fun run _ =
            RunCML.doit(main, SOME 20)
        end
      end
```

The second program is a demonstration of tightly coupled interaction between agents. The admissible program is discussed in section 5.2. The double and triple agents execute in parallel but they alternate in instantiating the tail of the admissible list. The messages from variables have been turned off to reduce the output. The messages **ask(true = true)** or **ask(false = true)** are generated by the guards of the triple agent Relational operators are implemented as functions that return a constraint. Their implied ask are not printed because they tend to generate a lot of output of the form **ask(Isint(*i*))**, where *i* is an integer constant. You can see from the tell requests of the from **tell(X=*i*)** that the values in the list are created in increasing order, demonstrating the alternation between the double and the triple agents.

```
% cat admissible

proc main (L)

  proc double (L)
  begin {
    ask(istree(cons3,3,L)) -> {
      new X, Y, L1 in {
            tell(L=cons3(X,Y,L1))
         || tell(Y=2*X)
         || double(L1)
      }
    }
  }
  end

  proc triple (L)
  begin {
    new X,Y,L1 in {
          tell(L=cons3(X,Y,L1))
       || ask(Y<30000) ->
              new Y1,Z,L2 in  {
                    tell(L1=cons3(Y1,Z,L2))
```

```
                || tell(Y1=3*Y)
                || triple(L1)
              }
          || ask(Y>=30000) ->
              tell(L1=nil)
        }
    }
    end

begin {
      double(L)
    || triple(L)
    || new S,U in tell(L=cons3(1,S,U))
}
end



% fg

- translate "admissible"
- use "admissible.ml"
[opening admissible.ml]
structure Prog :
  sig
    val main : unit -> unit
    val run : 'a -> unit
  end
[closing admissible.ml]

- Variable.print_trace := false;

- Prog.run ();
[9]:   ask(Istree(cons3,3,T1))
[6]:   tell(T1 = cons3(1, T2, T3))
[16]:  tell(T1 = cons3(T2, T3, T4))
[6]:   tell(T1 = cons3(1, T2, T3)): accepted
```

```
[9]:   ask(Istree(cons3,3,T1)): entailed
[9]:   ask(Istree(cons3,3,T1))
[16]:  tell(T1 = cons3(T2, T3, T4)): accepted
[21]:  tell(cons3(1, T1, T2) = cons3(T3, T4, T5))
[22]:  tell(T1 = 2)
[21]:  tell(cons3(1, T1, T2) = cons3(T3, T4, T5)): accepted
[22]:  tell(T1 = 2): accepted
[10]:  ask(false = true)
[17]:  ask(true = true)
[10]:  ask(false = true): inconsistent
[17]:  ask(true = true): entailed
[26]:  tell(T1 = cons3(T2, T3, T4))
[27]:  tell(T1 = 6)
[26]:  tell(T1 = cons3(T2, T3, T4)): accepted
[27]:  tell(T1 = 6): accepted
[9]:   ask(Istree(cons3,3,T1)): entailed
[31]:  tell(T1 = cons3(T2, T3, T4))
[9]:   ask(Istree(cons3,3,T1))
[36]:  tell(cons3(6, T1, T2) = cons3(T3, T4, T5))
[31]:  tell(T1 = cons3(T2, T3, T4)): accepted
[37]:  tell(T1 = 12)
[37]:  tell(T1 = 12): accepted
[36]:  tell(cons3(6, T1, T2) = cons3(T3, T4, T5)): accepted
[17]:  ask(false = true)
[17]:  ask(false = true): inconsistent
[32]:  ask(true = true)
[32]:  ask(true = true): entailed
[41]:  tell(T1 = cons3(T2, T3, T4))
[42]:  tell(T1 = 36)
[41]:  tell(T1 = cons3(T2, T3, T4)): accepted
[42]:  tell(T1 = 36): accepted
[9]:   ask(Istree(cons3,3,T1)): entailed
[46]:  tell(T1 = cons3(T2, T3, T4))
[9]:   ask(Istree(cons3,3,T1))
[51]:  tell(cons3(36, T1, T2) = cons3(T3, T4, T5))
[46]:  tell(T1 = cons3(T2, T3, T4)): accepted
[52]:  tell(T1 = 72)
[52]:  tell(T1 = 72): accepted
```

```
[51]:  tell(cons3(36, T1, T2) = cons3(T3, T4, T5)): accepted
[32]:  ask(false = true)
[47]:  ask(true = true)
[32]:  ask(false = true): inconsistent
[47]:  ask(true = true): entailed
[56]:  tell(T1 = cons3(T2, T3, T4))
[57]:  tell(T1 = 216)
[56]:  tell(T1 = cons3(T2, T3, T4)): accepted
[57]:  tell(T1 = 216): accepted
[9]:   ask(Istree(cons3,3,T1)): entailed
[61]:  tell(T1 = cons3(T2, T3, T4))
[9]:   ask(Istree(cons3,3,T1))
[66]:  tell(cons3(216, T1, T2) = cons3(T3, T4, T5))
[61]:  tell(T1 = cons3(T2, T3, T4)): accepted
[67]:  tell(T1 = 432)
[67]:  tell(T1 = 432): accepted
[66]:  tell(cons3(216, T1, T2) = cons3(T3, T4, T5)): accepted
[47]:  ask(false = true)
[62]:  ask(true = true)
[47]:  ask(false = true): inconsistent
[62]:  ask(true = true): entailed
[71]:  tell(T1 = cons3(T2, T3, T4))
[72]:  tell(T1 = 1296)
[71]:  tell(T1 = cons3(T2, T3, T4)): accepted
[72]:  tell(T1 = 1296): accepted
[9]:   ask(Istree(cons3,3,T1)): entailed
[76]:  tell(T1 = cons3(T2, T3, T4))
[9]:   ask(Istree(cons3,3,T1))
[81]:  tell(cons3(1296, T1, T2) = cons3(T3, T4, T5))
[76]:  tell(T1 = cons3(T2, T3, T4)): accepted
[82]:  tell(T1 = 2592)
[82]:  tell(T1 = 2592): accepted
[81]:  tell(cons3(1296, T1, T2) = cons3(T3, T4, T5)): accepted
[62]:  ask(false = true)
[77]:  ask(true = true)
[62]:  ask(false = true): inconsistent
[77]:  ask(true = true): entailed
[86]:  tell(T1 = cons3(T2, T3, T4))
```

```
[87]:  tell(T1 = 7776)
[86]:  tell(T1 = cons3(T2, T3, T4)): accepted
[87]:  tell(T1 = 7776): accepted
[9]:   ask(Istree(cons3,3,T1)): entailed
[91]:  tell(T1 = cons3(T2, T3, T4))
[9]:   ask(Istree(cons3,3,T1))
[96]:  tell(cons3(7776, T1, T2) = cons3(T3, T4, T5))
[91]:  tell(T1 = cons3(T2, T3, T4)): accepted
[97]:  tell(T1 = 15552)
[97]:  tell(T1 = 15552): accepted
[96]:  tell(cons3(7776, T1, T2) = cons3(T3, T4, T5)): accepted
[77]:  ask(false = true)
[92]:  ask(true = true)
[77]:  ask(false = true): inconsistent
[92]:  ask(true = true): entailed
[101]: tell(T1 = cons3(T2, T3, T4))
[102]: tell(T1 = 46656)
[101]: tell(T1 = cons3(T2, T3, T4)): accepted
[102]: tell(T1 = 46656): accepted
[9]:   ask(Istree(cons3,3,T1)): entailed
[106]: tell(T1 = cons3(T2, T3, T4))
[9]:   ask(Istree(cons3,3,T1))
[111]: tell(cons3(46656, T1, T2) = cons3(T3, T4, T5))
[106]: tell(T1 = cons3(T2, T3, T4)): accepted
[112]: tell(T1 = 93312)
[112]: tell(T1 = 93312): accepted
[111]: tell(cons3(46656, T1, T2) = cons3(T3, T4, T5)): accepted
[92]:  ask(true = true)
[107]: ask(false = true)
[92]:  ask(true = true): entailed
[107]: ask(false = true): inconsistent
[92]:  tell(T1 = nil)
[92]:  tell(T1 = nil): accepted
[9]:   ask(Istree(cons3,3,T1)): inconsistent

Answer:

L = cons3(1, 2, cons3(6, 12, cons3(36, 72, cons3(216, 432,
     cons3(1296, 2592, cons3(7776, 15552, cons3(46656, 93312, nil))))))
Ok




^Z
Stopped


% cat admissible.ml

structure Prog =
  struct
    local open Domain Solver in
      fun main () = let
        val L = new()

        fun double (L) = let
          in
            (
              ( ask(Istree("cons3",3,L));
                (
                  let
                    val X = new()
                    val Y = new()
                    val L1 = new()
                  in
                    (
                      fork (FN (fn () =>
                          tell((L==Tree("cons3",3,[X, Y, L1])))
                      ));
                      fork (FN (fn () =>
                          tell((Y==(Int(2)**X)))
                      ));
                      double(L1)
                    )
                  end
    )
   )
  )
 end

and triple (L) = let
 in
  (
   let
    val X = new()
    val Y = new()
    val L1 = new()
   in
    (
     fork (FN (fn () =>
       tell((L==Tree("cons3",3,[X, Y, L1])))
     ));
     fork (FN (fn () =>
       ( ask((Y<<Int(30000)));
        let
         val Y1 = new()
         val Z = new()
         val L2 = new()
        in
         (
          fork (FN (fn () =>
            tell((L1==Tree("cons3",3,[Y1, Z, L2])))
          ));
          fork (FN (fn () =>
            tell((Y1==(Int(3)**Y)))
          ));
          triple(L1)
         )
        end
       )
     ));
     ( ask((Y>=>=Int(30000)));
      tell((L1==Tree("nil",0,[])))
```

```
                    )
                  )
                end
              )
            end
        in
          answer (Names [(L,"L")]);
          (
            fork (FN (fn () =>
                double(L)
            ));
            fork (FN (fn () =>
                triple(L)
            ));
            let
              val S = new()
              val U = new()
            in
              tell((L==Tree("cons3",3,[Int(1), S, U])))
            end
          )
        end

        fun run _ =
          RunCML.doit(main, SOME 20)
      end
    end
```

Bibliography

[BL86] Marco Bellia and Giorgio Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3(3) 217 236, October 1986

[Bor79] A. H. Borning. *ThingLab --A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, March 1979. Xerox PARC Technical Report SSL 79-3.

[Bor81] A. H. Borning. The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[CG81] Keith Clark and Steve Gregory. A relational language for parallel programming. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171 178, October 1981 Reprinted in chapter 1 of [Sha87].

[CG86] K. L. Clark and S. Gregory. PARLOG: Parallel programming in logic *ACM Transactions on Programming Languages and Systems*, 8(1):1 49, January 1986 A revision appears in chapter 3 of [Sha87].

[Cla90] K. L. Clark. Parallel logic programming. *The Computer Journal*, 33(6):482 493, 1990.

[Coh90] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52-68, July 1990.

[Col82] Alain Colmerauer. Prolog and infinite terms. In K. A. Clark and S.-A. Tarnlund, editors, *Logic Programming*, APIC Studies in Data Processing No 16, pages 231–251. Academic Press, 1982

[Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees In *Proceedings of the 2nd International Conference on fifth Generation Computer Systems*, pages 85--99, Tokyo, November 1984.

[Col86] Alain Colmerauer Theoretical model of Prolog II. In M. van Caneghem and D. H. D. Warren, editors, *Logic Programming and its Applications*, chapter 1, pages 3 31 Ablex Pub., 1986

[Col89] Tom Colon. *Programming in PARLOG* International Series in Logic Programming. Addison-Wesley, 1989

[Deb91] Saumya K. Debray QD-Janus A prolog implementation of Janus. Unpublished note, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, debray@cs.arizona.edu, May 1991.

[DKM84] Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification *Journal of Logic Programming*, 1(1):35-50, 1984.

[DL86] Doug DeGroot and Gary Lindstrom, editors *Logic Programming, Functions Relations and Equations*. Prentice-Hall, 1986

[GM86] Joseph A Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295 363 Prentice-Hall, 1986

[Gre87] Steve Gregory. *Parrallel Logic Programming in PARLOG: The Language and Its Implementation.* International Series in Logic Programming. Addison-Wesley, 1987

[Haw91] David J. Hawley The concurrent constraint language GDCC and its parallel constraint solver. In *KL1 Workshop.* Institute for New Generation Computer Technology (ICOT), 1991.

[Hen89] Pascal Van Hentenryck *Constraint Satisfaction in Logic Programming.* MIT Press Series in Logic Programming. MIT Press, 1989.

[Her30] Jacques Herbrand Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovie, Class III Sci. Math. Phys* , 33, 1930.

[Hir86] Masahiro Hirata. Programming language Doc and its self-description, or, $X = X$ considered harmful. In *Proceedings of the 3rd Conference of Japan Society of Software Science and Technology*, pages 69-72, 1986

[HJM+87] Nevin Heintze, Joxan Jaffar. Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) programmer's manual, version 2.0 Technical report, Department of Computer Science, Monash University, Australia, 1987.

[Jaf84] Joxan Jaffar. Efficient unification over infinite terms *New Generation Computing*, 2:207 219, 1984.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages* pages 111 119, Munich, January 1987 Full paper in Technical Report 86/73, Department of Computer Science, Monash University, June 1986

[JLM86] Joxan Jaffar, Jean-Louis Lassez, and Michael J Maher A logic programming language scheme In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 441 467 Prentice Hall 1986.

[JM87] Joxan Jaffar and Spiro Michaylov Methodology and implementation of a CLP system In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 196 218 Melbourne, May 1987 MIT Press.

[Kni89] Kevin Knight Unification. a multidisciplinary survey *ACM Computing Surveys*, 21(1):93 124, March 1989.

[LBD$^+$88] E. Lusk, R. Butler, T Disz, R Olson, R Overbeek, R Stevens, D H D Warren A. Calderwood, P Szeredi, S Haridi, P Brand, M Carlsson, A Ciepielewski and B. Hausman The aurora or-parallel prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819 830, ICOT, Tokyo, 1988

[Lel88] Wm Leler *Constraint Programming Languages, Their Specification and Generation.* Addison-Wesley, 1988.

[LMM88] Jean-Louis Lassez, Michael J. Maher, and K. Marriot. Unification revisited In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587 625 Morgan Kaufman, 1988

[LS90] Pierre Lim and Peter J Stuckey A constraint logic programming shell. In P. Deransart and J. Maluszynski, editors, *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 456 pages 75-88, Linkoping, Sweden, 1990 Springer-Verlag

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm *ACM Trancations on Programming Languages and Systems*, 4(2) 258 282, April 1982

[MR84] Alberto Martelli and Gianfranco Rossi. Efficient unification with infinite terms in logic programming. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 202–209, 1984.

[Mt91] Robin Milner and Mads tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MtH90] Robin Milner, Mads tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nai86] Lee Naish. *Negation and Control in Prolog*. Lecture Notes in Computer Science 238. Springer-Verlag, 1986.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rep90] John H. Reppy. *Concurrent Programming with Events - The Concurrent ML Manual*, November 1990. Updated for Version 0.9.5, July 1991.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, Canada, June 1991. published as SIGPLAN Notices 26(6).

[Rep92] John H. Reppy. *Higher-order concurrency*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, January 1992. forthcoming.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To appear in Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1991.

[Sha87] Ehud Shapiro, editor. *Concurrent Prolog, Collected Papers*, volume 1 & 2 of *MIT Press Series in Logic Programming*. MIT Press, 1987.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[SKL89] Vijay A. Saraswat, Ken Khan, and Jacob Levy. Programming in Janus. Technical report, Xerox PARC, December 1989.

[SKL90] Vijay A. Saraswat, Ken Khan, and Jacob Levy. JANUS: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, Austin, Texas, October 1990. Full paper is a technical report from XeroX PARC, December 1989.

[SR90] Vijay A Saraswat and Martin Rinard Concurrent constraint programming In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 232 241, San Francisco, January 1990

[SRP91] Vijay A Saraswat, Martin Rinard, and Prakash Panangaden Semantic foundations of concurrent constraint programming In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991

[Ste80] Guy L. Steele *The Definition and Implementation of a Computer Programming Language Based on Constraints* PhD thesis, MIT, Cambridge, MA, August 1980. Technical Report MIT-AI TR 595

[Sut63] Ivan Sutherland *Sketchpad A man-machine graphical communication system* Outstanding Dissertations in Computer Science Garland Publishing Inc , 1963

[SY86] P A. Subrahmanyam and Jia-Hua You FUNLOG a computational model integrating logic programming and functional programming In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 157 198 Prentice-Hall 1986

[Tay88] H. Taylor Localizing the GHC suspension test In K Bowen and R.A Kowalski editors, *Proceedings of the fifth International Conference Symposium on Logic Programming*, pages 1257 1271 MIT Press, 1988

[Ued86a] Kazunori Ueda Guarded horn clauses In E Wada, editor, *Logic Programming 1985*, Lecture Notes in Computer Science 221, pages 168 179, 1986 A revision appears in chapter 4 of [Sha87]

[Ued86b] Kazunori Ueda *Guarded Horn Clauses* PhD thesis, University of Tokyo, 1986

[vEdLF82] M. H. van Emden and G J de Lucena Filho. Predicate logic as a language for parallel programming In K. A. Clark and S -A Tarnlund, editors, *Logic Programming*, APIC Studies in Data Processing No 16, pages 189 198 Academic Press, 1982.

[Yas84] Hiroto Yasuura On parallel computational complexity of unification In *Proceedings of the 2nd International Conference on fifth Generation Computer Systems* pages 235 243, Tokyo, November 1984.