First-order Lax Logic as a Framework for Constraint Logic Programming

Matt Fairtlough* Michael Mendler† Matt Walton*

December 1, 1997

Abstract

In this report we introduce a new proof-theoretic approach to the semantics of Constraint Logic Programming, based on an intuitionistic first-order modal logic, called QLL. The distinguishing feature of this new approach is that the logic calculus of QLL is used not only to capture the usual *extensional* aspects of Logic Programming, *i.e.* "which queries are successful," but also some of the *intensional aspects*, *i.e.* "what is the answer constraint and how is it constructed." It provides for a direct link between the model-theoretic and the operational semantics following a *formulas-as-programs* and *proofs-as-constraints* principle.

This approach makes use of logic in a different way than other approaches based on logic calculi. On the one side it is to be distinguished from the well-known *provability semantics* which is concerned merely with *what* is derivable as opposed to *how* it is derivable, paying attention to the fact that it is the *how* that determines the answer constraint. On the other side our approach is distinguished from so-called *external* logic characterizations of the operational semantics of logic programming. There, operational semantics are axiomatized in a classical meta-logic specifying the program behaviour in terms of successful and failing queries. Here, in contrast, QLL is used as an *internal* logic in which operational semantics are not differentiated by different axiom systems but by different rule systems. Or, to put it at another level: Formulas in QLL do not specify properties of programs but the programs themselves.

Technically, we first extend existing work on Propositional Lax Logic (PLL) to a first-order language (QLL), presenting a soundness and completeness theorem for a Gentzen-style system via a syntactic translation into a classical first-order bimodal theory. Previous work on applying Lax Logic to deal with behavioural constraints in formal hardware verification has demonstrated the complementary nature of abstraction and constraints; we proceed to show how the Lax Logic Programming (LLP) fragment of QLL can reveal abstractions behind Constraint Logic Programming (CLP). Our main tool is an *intensional* first-order extension of the Curry-Howard isomorphism between natural deduction proofs in PLL and terms of the computational lambda calculus. Instantiating the monadic operator of QLL by a generic notion of constraint computation, we factor a concrete CLP program into two parts: an abstract LLP program and an associated constraint table. These tables allow us to recover concrete answer constraints for CLP programs from abstract LLP derivations, and thus to establish precise proof- and model-theoretic connections between our Lax logical account of CLP and existing work. Choosing different notions of constraint allows us to generalize the standard notion of constraint (as in CLP) and to apply the LLP paradigm to the complementary problems of program abstraction and program refinement.

*Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello St., Sheffield S1 4DP. Emails: `m.fairtlough@dcs.shef.ac.uk`, `m.walton@dcs.shef.ac.uk`

†Department of Mathematics and Computer Science, University of Passau, Innstraße 33, D-94032 Passau. Email: `mendler@fmi.uni-passau.de` The author is supported by the *Deutsche Forschungsgemeinschaft*.

Contents

1 Introduction

This report investigates Quantified Lax Logic (QLL), a first-order extension of Propositional Lax Logic [FM95, FM97], and its application to the analysis of constraints. Our working definition is that a constraint is a property of a system's context which validates a specified abstract model of its behaviour; for us, *abstraction* and *constraints* are thus two sides of the same coin. Indeed, previous work on applying Lax Logic [Men90, Men93, MF96, Men96] to deal with behavioural constraints in formal hardware verification suggests a basic principle:

> *"Every design abstraction has a corresponding calculus of constraints, and every use of constraints corresponds to some design abstraction."*

This is a rather dedicated view of the notion of a constraint that contrasts with the traditional

> *constraints-as-built-in-predicates*

paradigm of Constraint Logic Programming (CLP). By exploring the Lax Logic Programming (LLP) fragment of QLL in this work we try to convince the reader of the benefits of linking constraints with abstractions, while taking both as equally important. There is a remarkable body of previous work on abstractions in theorem proving, see *e.g.* [Pla81, GW92], which does not mention the concept of constraints, while the work on constraint logic programming does not seem to mention abstractions, see *e.g.* [JM94].

Before we tackle the technicalities some general remarks may be in order to set this work in a wider context. The problem of context (or environment) arises in most if not all of our attempts to understand our surroundings. For example, see [Sch89] for a highly entertaining account of the problem in philosophy and anthropology.

In science, the problem becomes sharper, for virtually all scientific investigations introduce some form of abstraction to deal with the enormous complexity presented to us by the world. The constraints under which the abstraction is valid are not always explicitly acknowledged or even understood, although they must always be present in some form. An obvious example is Newtonian mechanics, where the abstract model of behaviour is accurate provided the speeds and gravitational fields involved are sufficiently small. The model may be refined to a more accurate relativistic form, which would itself embody a refined notion of constraint ... Similar hierarchical chains of abstractions, concerning data, timing, and structural aspects, frequently arise in hard- or software verification. Each abstraction has an associated notion of constraint to be imposed on the context, introducing a distinct compositionality problem. It is evident that the problem has no definitive solution, for in the limit the context of a system is the rest of the universe including its past and future evolution, and we have no hope of understanding that.

Having said that, in situations where the relevant constraints have been identified and aspects of system behaviour can be adequately formalised in logic, it seems worthwhile to try to provide a general theory and a formal mechanism for handling the awkward passage across abstraction boundaries—indeed it may well be that most errors in system verification creep in at these boundaries. Lax Logic is such a general theory. It is an intuitionistic approach that relaxes the classical notion of absolute correctness to rely instead on relative correctness *up-to* constraints, which is formalised, in a strictly deductive sense, as a logical modality $\bigcirc$. Such a modality must be intrinsically intuitionistic[1] for if $\bigcirc M$ is to mean that "M is true up to constraints" this must hold on the basis of only partial information about the context. Truth must be sustainable under specialisation of the context, and the information we can positively rely on is what can be deduced from the structure of the formula M itself. Precisely this is a characteristic feature of intuitionistic logic.

[1] or constructive for that matter, but we take intuitionistic logic as the best available, in the sense of being sufficiently incontrovertible, approximation to the constructive principle.

In contrast, classical logic has a closed world assumption to justify indirect arguments deducing the presence of some features from the absence of others. Classical correctness, if taken strictly, assumes complete knowledge of the context of a system and the interaction of all of its components into all levels of detail.

In this report we explore the connection between abstraction and constraints as it applies to CLP and show how CLP naturally corresponds to the Lax Logic Programming (LLP) fragment of QLL. The calculus of QLL formalises the generic properties of $\bigcirc$, while any concrete notion of constraint arises from a concrete semantic interpretation of QLL. Technically, our claim

notion of constraint = *semantic interpretation of* QLL

has a *proof-theoretic* and a *model-theoretic* side. Proof-theoretically we specialise this to the identification of a notion of constraint with a model of the *computational lambda calculus* [Mog91] exploiting the Curry-Howard isomorphism for QLL. Model-theoretically our claim specialises to the identification of a notion of constraint with a (class of) *Kripke constraint models*. To fit the structure of QLL we extend the computational lambda calculus by dependent types and the Kripke constraint models [FM95, FM97] by universes. For Kripke constraint models we give soundness and completeness results. We further present a specific family of computational lambda calculi and Kripke constraint models, so that LLP for these semantics captures the essence of CLP. The parameter of these families corresponds to the different choices of the constraint domain in the CLP($\mathcal{X}$) [JM94] scheme. The LLP paradigm provides a uniform framework for CLP in which the intensional separation between programs and constraints is captured in a rather natural way: Programs are formulas and constraints are (abstract interpretations of) proofs. By choosing different constraint semantics for proofs we can generalise the standard notion of constraint and thus accommodate the complementary concepts of program abstraction and program refinement.

The main tool of this work is the computational lambda calculus (λ_c). It was introduced originally by Moggi [Mog88] as a coherent algebraic model for partial functions. In subsequent work Moggi observed that this algebraic structure could be seen as an extension of simply typed λ-calculus by a strong monad $\bigcirc$, and that this description not only encompasses partial functions but a variety of other *notions of computation*, such as nondeterminism, side-effects, continuations, *etc.* [Mog90, Mog91]. Since then the notion of a (strong) monad has been used very successfully in functional programming, *e.g.* by Wadler [Wad90, Wad92], and even incorporated into the programming language Haskell [Tho96]. While monads are now well established in functional programming as a structuring principle and mechanism for semantic extensions, the relevance of λ_c as a calculus of proofs, *quā* the Curry-Howard isomorphism, is still largely unexplored. Some theoretical work has been done *e.g.* by Benton *et al.* [BBdP95], where the logic of $\bigcirc$ is called 'computational logic.' The practical potential of λ_c for formal system verification, on the other hand, is indicated in a series of publications [Men90, MF96, Men96] where the monad $\bigcirc$ shows up as a *notion of constraint.* It is this interpretation that we wish to advance further by the results in this report, offering it as a fruitful, so we believe, logic pendant to Moggi's original idea.

2 Constraint Logic Programming CLP

Constraint Logic Programming (CLP) is the result of a merger between two declarative paradigms — constraint solving and logic programming ([FHK$^+$92], [Kri92], [Las87], [Pou95] are suitable introductions). In [JL87] Jaffar and Lassez propose a general scheme, CLP($\mathcal{X}$); a logic programming framework which replaces unification with constraint solving in the domain of choice, $\mathcal{X}$. In fact, unification can be seen as a special instance of constraint solving, specifically over the Herbrand Universe.

The result is a scheme for CLP languages which possess greatly increased expressive power. Several languages based on CLP($\mathcal{X}$) have already been implemented, each over specific constraint domains. For example, CLP($\mathcal{R}$) ([JMSY92]) computes linear arithmetic constraints over the real numbers and Prolog III ([Col87, Col90]) computes over two-valued Boolean logic, lists and linear arithmetic over rational numbers. ECLiPSe ([MSea93], a successor to the CHIP system [DvHSA88]) computes constraints over an extended Boolean algebra with symbolic constraints and performs linear arithmetic over both rational numbers and bounded subsets of integers (known as "finite domains"). For further accounts of these and many other such languages see [Coh90] and [JM94].

Theoretically, CLP preserves and extends the semantic properties of pure logic programs, such as operational, logical and fix-point semantics which coincide in a natural way (compare, for example, [Llo87] with [JMMS96]). Indeed, it is the operational semantics with which we are interested in this report and these will be studied in greater detail later. The following CLP example, which is taken from [Kri92], illustrates how constraints can be combined into the context of logic programs to provide a succinct and expressive declarative language.

Example 2.1 *In this* CLP($\mathcal{R}$) *program,* `mortgage` *is defined in terms of the principal* `P`*, duration* `D`*, interest rate* `I`*, monthly payments* `MP` *and balance* `B`*.*

```
mortgage(P, I, MP, B, D) :-
      D <= 1,
      B+MP = P*(I+1).
mortgage(P, I, MP, B, D) :-
      1 < D,
      mortgage(P*(I+1)-MP, I, MP, B, D-1).
```

The system can be used to fully evaluate a goal for a particular variable, e.g.

```
?- D = 120, I = 0.01, B = 0, MP = 1721.65, mortgage(P, I, MP, B, D).
```

yields the result `P = 120000`*. However, given a goal which specifies values for just* `D`*,* `I` *and* `B`*, the system performs partial evaluation for* `P` *and* `MP`*, e.g.*

```
?- D = 5, I = 0.01, B = 0, mortgage(P, I, MP, B, D).
```

yields the result `MP = 0.263797522*P`*.*

The expressive power stems from the fact that by using built-in constraint predicates, such as in the predicate call `B+MP=P*(I+1)` of the example above the user program can invoke specialised and pre-compiled decision procedures. This way the programmer bypasses the limitations defined by the abstractness of the logic programming language and reaches through to more concrete semantic levels. These semantic levels are outside of the logic programming model and their operational details are typically abstracted away by referring to a model-theoretic account of constraint relations. Indeed, according to the standard semantic view [JM94] CLP($\mathcal{X}$) is explained as a parametrisation of the semantics of LP (be it operational, declarative, or fixed-point) in a model-theoretic semantics of the domain of computation and constraints $\mathcal{X}$.

It has been noted, however, that this straightforward approach makes for a rather weak link between constraints and ordinary logic programming, which does not adequately capture a number of operational aspects of this relationship. For one, the fixed-point semantics no longer corresponds to computations. Since it refers to the model-theory of $\mathcal{X}$ it relies on omnipotent semantic reasoning, rather than formal and effective deduction of constraints. As pointed out by Argenius and Voronkov [AV96] it is possible to hide an arbitrary amount of computations in the constraint level, which is then available within the very first iteration of the fixed-point model. Secondly, it does not capture the essential difference between constraints and propositions: the semantically relevant information in user predicates merely is whether or not they are consequences of a program (`yes/no`); answer constraints, in contrast, are relevant not because they are derivable from some syntactic or semantic theory, but because they are symbolic representations of a solution set (`MP = 0.263797522*P`). Thus, constraints have an intensional flavour that needs to be represented by an adequate semantics for CLP. For LP a more intensional semantics of answer constraints, called *s-semantics* has been proposed [FLMP89] to capture this aspect, for the special class of unification constraints.

In this work we present a new proof-theoretic approach to CLP which is even more intensional than the *s-semantics*. It provides for a strong link between the operational semantics of LP and constraint generation, and it does not depend on a model-theoretic semantics for constraints. We propose QLL as a logic framework in which CLP programs are a fragment of formulas and operational semantics for CLP correspond to derived logic calculi for this fragment. The general philosophy behind this proof-theoretic approach is this:

> Given that a logic calculus, especially a constructive one, is a good way to formalise intensional aspects of a logic theory, and if logic programs truly can be seen as logic theories, then it should also be possible to capture the intensional aspects related to the execution of logic programs in terms of logic calculi.

The goal of treating operational semantics of logic programs as logic proof rules may appear mistaken. Given the intrinsic mismatch between constraint verification and constraint generation one may be lead to conclude that "logic programming systems are not in general theorem-provers" [HSH90]. In this report we show they are, provided we take the right abstract point if view.

Proof-theoretically, the execution of a query (open formula) `mortgage(P, 0.01, MP, 0, 5)` with respect to a program `mortgage` is equivalent to proving the existential closure $\exists \mathtt{P},\mathtt{MP} : \mathbf{U}.\, \mathtt{mortgage}(\mathtt{P}, 0.01, \mathtt{MP}, 0, 5)$, where $\mathbf{U}$ stands for the universe, in the logic calculus related to `mortgage`. Provided the calculus is constructive this corresponds to exhibiting a satisfying substitution for `mortgage(P, 0.01, MP, 0, 5)`. This seems easy enough but there still is some way to go from knowing that an answer constraint exists to actually computing it, and moreover computing it in as general a form as possible. In fact, if all we have is the standard existential property of constructive logic we are only entitled to infer the existence[2] of a *ground* substitution for `mortgage(P, 0.01, MP, 0, 5)` which is hardly adequate for logic programming. It seems we need stronger constructive principles to explain the generation of answer constraints. In this report, we solve this proof-theoretic problem in the following form: We take the execution of the query `mortgage(P, 0.01, MP, 0, 5)` to be essentially equivalent to proving a $\forall\exists$ formula

$$\forall \mathtt{P}, \mathtt{MP}.\, \exists \mathtt{c}.\, \mathtt{c} \supset \mathtt{mortgage}(\mathtt{P}, 0.01, \mathtt{MP}, 0, 5)$$

where the existentially quantified `c` ranges over a class $\mathbf{C}$ of constraints, and $\supset$ is logic implication. Now, if the logic is constructive this implies the existence of a closed function term $\mathtt{C} : (\mathbf{U} \times \mathbf{U}) \Rightarrow \mathbf{C}$ such that

$$\forall \mathtt{P}, \mathtt{MP}.\, \mathtt{C}(\mathtt{P}, \mathtt{MP}) \supset \mathtt{mortgage}(\mathtt{P}, 0.01, \mathtt{MP}, 0, 5).$$

The term `C` is the answer constraint, and if things are set up symbolically enough, then we may have $\mathtt{C}(\mathtt{P}, \mathtt{MP}) = (\mathtt{MP} = 0.263797522 * \mathtt{P})$ as desired. To refine the picture further, we note that we can do without the second-order existential quantifier over constraints. Since we only ever need the

[2] As pointed out in [HSH90], the existence of a ground answer substitution is guaranteed already by the soundness of logic programming for Herbrand models. So, the constructive nature of the calculus does not add any extra information.

quantifier in the special form $\exists \mathtt{c} : \mathbf{C}.\, \mathtt{c} \supset \mathtt{S}$ where $\mathtt{S}$ is a (query) formula we may as well hard-wire it in and replace it by a new modal operator $\bigcirc\mathtt{S}$. Then the query becomes

$$\forall \mathtt{P}, \mathtt{MP}. \bigcirc \mathtt{mortgage}(\mathtt{P}, \mathtt{0.01}, \mathtt{MP}, 0, 5).$$

This, in a nutshell, motivates our interest in a first-order intuitionistic modal logic. Also observe how the operational semantics of CLP is captured by this constructive proof-theoretic way of arranging affairs: Since the extracted answer constraint depends on the proof, different proofs will give rise to different constraints. This suggests that particular operational semantics for CLP, considered as methods of constructing constraints, may be characterised by particular logic calculi and proof search strategies for QLL. Also, this implies that in order to get *complete* constraint information for a given query $\mathtt{mortgage}(\mathtt{P}, \mathtt{0.01}, \mathtt{MP}, 0, 5)$ we must consider the set of *all* proofs of $\forall \mathtt{P}, \mathtt{MP}. \bigcirc \mathtt{mortgage}(\mathtt{P}, \mathtt{0.01}, \mathtt{MP}, 0, 5)$. This matches the well-known result that operational completeness for CLP, in general, requires to collect the answer constraints of several successful executions (see *e.g.* [JM94]). In order not to claim too much it should be added that in this report we do not analyse such a collection process.

3 Quantified Lax Logic QLL

Here we introduce the first-order extension to Propositional Lax Logic (PLL) [FM95, FM97], dubbed QLL. Its semantics is given in terms of Kripke-style models and there are complete proof systems in the form of both Gentzen and natural deduction sequents, and Hilbert deduction. In this section we only briefly touch upon the Gentzen system, the natural deduction calculus introduced in Sec. 4 being our main concern. The Hilbert axiomatisation is obtained simply as a first-order intuitionistic axiomatic extension of the Hilbert system for PLL.

We have a language $\mathcal{L}$ of predicate symbols P and first-order terms t. The terms are constructed in the usual manner from variables x, constants c and function symbols f.

Definition 3.1 *Formulas M of QLL are generated by the following grammar*

$$M \quad ::= \quad P \mid \mathit{false} \mid M \wedge M \mid M \vee M \mid M \supset M \mid \bigcirc M \mid \forall x.\, M(x) \mid \exists x.\, M(x)$$

where P is a meta-variable for atomic formulas $P_n(t_1, \ldots, t_n)$. Negated formulas may be defined by $M \supset \mathit{false}$.

The model theory for QLL based on a variant of Kripke models, called *constraint models*, is given in Section 8. Here we are more interested in the proof-theory of QLL, which is not only adequate for the constraint models but in addition captures some of the intensional semantics of constraint handling.

A multi-succedent Gentzen-style calculus for QLL is obtained from a first-order intuitionistic Gentzen sequent calculus (see [van89]) augmented by two Lax Logic-specific rules, $\bigcirc$-L and $\bigcirc$-R. This calculus is seen in Fig. 1. Sequents are of form $\Gamma \vdash \Delta$ where Γ and Δ are finite lists of formulas. We write $\vdash_{\mathrm{QLL}} M$ to denote that the sequent $\vdash M$ is derivable in QLL.

Theorem 3.2 (Cut-Elimination) *The cut rule Cut is admissible in QLL, i.e. if $\Gamma \vdash_{\mathrm{QLL}} M$, then this sequent is derivable without the Cut rule.*

A detailed proof of Cut-elimination can be found in [FW97]. It will sometimes be useful to extend QLL by particular theories, for instance to axiomatise constraints. We write QLL($\mathcal{T}$) for the extension of QLL by theory $\mathcal{T}$, which need not be purely axiomatic, but include extra rules as well. Specifically, QLL(=) is QLL extended by the usual equational reasoning.

Structural Rules

$$\frac{}{M \vdash M}\ \text{Axiom} \qquad \frac{\Gamma \vdash M, \Delta \qquad \Gamma, M \vdash \Delta}{\Gamma \vdash \Delta}\ \text{Cut}$$

$$\frac{\Gamma, M, N, \Gamma' \vdash \Delta}{\Gamma, N, M, \Gamma' \vdash \Delta}\ \text{ExchangeL} \qquad \frac{\Gamma \vdash \Delta, M, N, \Delta'}{\Gamma \vdash \Delta, N, M, \Delta'}\ \text{ExchangeR}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, M \vdash \Delta}\ \text{WeakL} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash M, \Delta}\ \text{WeakR}$$

$$\frac{\Gamma, M, M \vdash \Delta}{\Gamma, M \vdash \Delta}\ \text{ContractionL} \qquad \frac{\Gamma \vdash M, M, \Delta}{\Gamma \vdash M, \Delta}\ \text{ContractionR}$$

Logical Rules

$$\frac{}{\mathit{false} \vdash N}\ \text{FalseIntro}$$

$$\frac{\Gamma, M, N \vdash \Delta}{\Gamma, M \wedge N \vdash \Delta}\ \wedge\text{-L} \qquad \frac{\Gamma \vdash M, \Delta \qquad \Gamma \vdash N, \Delta}{\Gamma \vdash M \wedge N, \Delta}\ \wedge\text{-R}$$

$$\frac{\Gamma, M \vdash \Delta \qquad \Gamma, N \vdash \Delta}{\Gamma, M \vee N \vdash \Delta}\ \vee\text{-L} \qquad \frac{\Gamma \vdash M, N, \Delta}{\Gamma \vdash M \vee N, \Delta}\ \vee\text{-R}$$

$$\frac{\Gamma \vdash M, \Delta \qquad \Gamma, N \vdash \Delta}{\Gamma, M \supset N \vdash \Delta}\ \supset\text{-L} \qquad \frac{\Gamma, M \vdash N}{\Gamma \vdash M \supset N}\ \supset\text{-R}$$

$$\frac{\Gamma, M \vdash \bigcirc N}{\Gamma, \bigcirc M \vdash \bigcirc N}\ \bigcirc\text{-L} \qquad \frac{\Gamma \vdash M, \Delta}{\Gamma \vdash \bigcirc M, \Delta}\ \bigcirc\text{-R}$$

$$\frac{\Gamma, M(a) \vdash \Delta}{\Gamma, \exists x.M(x) \vdash \Delta}\ \exists\text{-L} \qquad \frac{\Gamma \vdash M(t), \Delta}{\Gamma \vdash \exists x.M(x), \Delta}\ \exists\text{-R}$$

$$\frac{\Gamma, M(t) \vdash \Delta}{\Gamma, \forall x.M(x) \vdash \Delta}\ \forall\text{-L} \qquad \frac{\Gamma \vdash M(a)}{\Gamma \vdash \forall x.M(x)}\ \forall\text{-R}$$

Restriction for rules $\forall$-R and $\exists$-L:
a must not occur in Γ, Δ or $M(x)$.

Figure 1: Gentzen Sequent Calculus Rules for QLL

4 Computational Lambda-Calculus $\lambda_c^{\Sigma\Pi}$ and Constraints

We begin our investigation into the connection between QLL and CLP by looking at a Natural Deduction presentation of QLL with its corresponding lambda-calculus $\lambda_c^{\Sigma\Pi}$ of proof terms. The rules of $\lambda_c^{\Sigma\Pi}$, which we refer to as QLL-ND, are given in Fig. 2. It is convenient to count *true* as an atomic formula and $*$ as a basic value, although they could both be defined using *false*. Let $\mathcal{U}$ be the set of well-formed object level terms, possibly containing free variables. In all rules mentioning object level terms t we suppose the terms are well-formed, *i.e.* $t \in \mathcal{U}$. If $\Gamma = M_1, \ldots, M_n$ we write $\Gamma \vdash_{\text{ND-QLL}} N$ or simply $\Gamma \vdash N$ to indicate that there is a proof of N from assumptions Γ in QLL-ND, *i.e.* that there is a proof term p and proof variables w_i such that $w_1 : M_1, \ldots, w_n : M_n \vdash p : N$.

Theorem 4.1 (Extensional equivalence) QLL-ND *is extensionally equivalent to the Gentzen system, in the sense that* $\Gamma \vdash_{\text{QLL}} N \iff \Gamma \vdash_{\text{ND-QLL}} N$ *for all formulas* N.

$$\frac{}{\Gamma, z : M, \Gamma' \vdash z : M}\ \mathcal{I} \qquad \frac{}{\Gamma \vdash * : \mathit{true}}\ \mathit{true}_{\mathcal{I}} \qquad \frac{\Gamma \vdash p : \mathit{false}}{\Gamma \vdash \mathsf{efq}(p) : M}\ \mathit{false}_{\mathcal{E}}$$

$$\frac{\Gamma \vdash p : M \qquad \Gamma \vdash q : N}{\Gamma \vdash (p, q) : M \wedge N}\ \wedge_{\mathcal{I}} \qquad \frac{\Gamma \vdash r : M \wedge N}{\Gamma \vdash \pi_1(r) : M}\ \wedge_{\mathcal{E}} \qquad \frac{\Gamma \vdash r : M \wedge N}{\Gamma \vdash \pi_2(r) : N}\ \wedge_{\mathcal{E}}$$

$$\frac{\Gamma \vdash r : M \vee N \qquad \Gamma, y : M \vdash p : K \qquad \Gamma, z : N \vdash q : K}{\Gamma \vdash \mathsf{case}\ r\ \mathsf{of}\ [\iota_1(y) \to p,\ \iota_2(z) \to q] : K}\ \vee_{\mathcal{E}}$$

$$\frac{\Gamma \vdash p : M}{\Gamma \vdash \iota_1(p) : M \vee N}\ \vee_{\mathcal{I}} \qquad \frac{\Gamma \vdash p : N}{\Gamma \vdash \iota_2(p) : M \vee N}\ \vee_{\mathcal{I}}$$

$$\frac{\Gamma, z : M \vdash p : N}{\Gamma \vdash \lambda z\,.\,p : M \supset N}\ \supset_{\mathcal{I}} \qquad \frac{\Gamma \vdash p : M \supset N \qquad \Gamma \vdash q : M}{\Gamma \vdash p\, q : N}\ \supset_{\mathcal{E}}$$

$$\frac{\Gamma \vdash p : M}{\Gamma \vdash \mathsf{val}(p) : \bigcirc M}\ \bigcirc_{\mathcal{I}} \qquad \frac{\Gamma \vdash p : \bigcirc M \qquad \Gamma, z : M \vdash q : \bigcirc N}{\Gamma \vdash \mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ q : \bigcirc N}\ \bigcirc_{\mathcal{E}}$$

$$\frac{\Gamma \vdash p : M}{\Gamma \vdash \langle p \mid x \rangle : \forall x.M}\ \forall_{\mathcal{I}} \quad (x \text{ not free in } \Gamma) \qquad \frac{\Gamma \vdash p : \forall x.M}{\Gamma \vdash \pi_t\, p : M[t/x]}\ \forall_{\mathcal{E}}$$

$$\frac{\Gamma \vdash p : M[t/x]}{\Gamma \vdash \iota_t(p) : \exists x.M}\ \exists_{\mathcal{I}}$$

$$\frac{\Gamma \vdash r : \exists x.M \qquad \Gamma, z : M \vdash p : K}{\Gamma \vdash \mathsf{case}\ r\ \mathsf{of}\ [\iota_x(z) \to p] : K}\ \exists_{\mathcal{E}} \quad (x \text{ not free in } K \text{ or } \Gamma)$$

Figure 2: Natural Deduction Rules for QLL.

Under the Curry-Howard isomorphism, or propositions-as-types principle, the rules of QLL-ND can be looked at from two directions: They can be viewed as typing rules for a formal extension of Moggi's Computational Lambda-calculus λ_c [Mog91] by first-order dependent sums Σ and products Π, corresponding to first-order existential and universal quantification respectively. They can also be viewed as proof assignment rules for QLL associating explicit proof terms to theorems of QLL. It is this second interpretation that we are interested in here. For us proofs represent extra *intensional* information about *why* a formula is true rather than simply the *extensional* fact *that* it is true.

As it turns out this extra intensional nature of proofs is sufficiently rich to accommodate what is referred to as a constraint both in hardware or software specification and in CLP programming. Our idea is that notions of constraint correspond to specific computational semantics for $\lambda_c^{\Sigma\Pi}$ proof terms such that we can interpret a proof p of a formula $\bigcirc M$ as the computation of a constraint c such that "M holds under c." More generally, each proof p of an arbitrary formula M computes some constraint information c, the exact nature of which depends on the structure of M, such that the derivation of $p : M$ corresponds to the statement "M-refined-by-c is true."

Let us postpone the relation between proofs and formulas for a short while and instead, for the remainder of this section, concentrate on the proof terms themselves and their computational interpretation. The proof terms generated by the QLL-ND rules are built according to the abstract grammar seen in Fig. 3. A proof term is *well-typed* if it can be typed by the QLL-ND rules of Fig. 2.

$$
\begin{array}{rcl}
p & ::= & z \mid * \mid \mathsf{efq}(p) \mid (p,p) \mid \pi_1(p) \mid \pi_2(p) \mid \\
 & & \mathsf{case}\ p\ \mathsf{of}\ [\iota_1(z_1) \to p,\ \iota_2(z_2) \to p] \mid \iota_1(p) \mid \iota_2(p) \mid \\
 & & \lambda z\,.\,p \mid p\,p \mid \mathsf{val}(p) \mid \mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ p \mid \\
 & & \langle p \mid x \rangle \mid \pi_t\,p \mid \iota_t(p) \mid \mathsf{case}\ p\ \mathsf{of}\ [\iota_x(z) \to p]
\end{array}
$$

x ranges over object variables, z, z_1, z_2 over proof variables, t over well-formed object-level terms.

Figure 3: Abstract Syntax of $\lambda_c^{\Sigma\Pi}$ Proofs.

As a very general computational interpretation of $\lambda_c^{\Sigma\Pi}$, obtaining a generic calculus of constraints, we take the equational theory induced by extending λ_c canonically by first-order dependent sums and products. This equational theory can be derived from the categorical models for Lax Logic, presented in [Men93], which are essentially hyper-doctrines with a strong monad. The β and η equations are given in Fig. 4.
We assume that the terms on both sides of every equation in Fig. 4 are well-typed. This implies, for instance, that in equation $\bigcirc$.ass variable y does not occur free in r. Apart from β and η equations we also adopt the usual reflexivity, symmetry, transitivity, and substitution rules for $=$. Furthermore, we include all the ξ rules to make $=$ a congruence also with respect to the variable binding operators $\mathsf{case}\ r\ \mathsf{of}\ [\iota_1(\underline{y}) \to p,\ \iota_2(\underline{z}) \to q]$, $\lambda\underline{z}\,.\,p$, $\mathsf{let}\ \underline{z} \Leftarrow p\ \mathsf{in}\ q$, $\langle p \mid \underline{x} \rangle$, $\mathsf{case}\ r\ \mathsf{of}\ [\iota_{\underline{x}}(\underline{z}) \to p]$, where the bound variables in each case are underlined. Finally, as usual, we identify terms up to α-conversion and assume that substitution $p\{t/x\}$ renames bound variables to avoid name capture.

Valuable as it would be to do so, we do not propose to investigate the equational theory or the general category-theoretic framework for $\lambda_c^{\Sigma\Pi}$ here; nor does space permit detailed discussion of reduction properties of $\lambda_c^{\Sigma\Pi}$. A purely propositional extension of Moggi's λ_c including sums has been analysed in [BBdP95] and proved to be strongly normalising under $\to_{\beta c}$ reduction[3]. Their paper is unusual in that λ_c, as here $\lambda_c^{\Sigma\Pi}$, is studied from a proof-theoretic point of view. This report takes [BBdP95] further both in extending λ_c to $\lambda_c^{\Sigma\Pi}$ and in specialising $\lambda_c^{\Sigma\Pi}$ from a calculus of proofs to a method for calculating constraints. In the next section we turn $\lambda_c^{\Sigma\Pi}$ into a more specific calculus of constraints, $\lambda_c(\mathcal{C})$, generated from a class $\mathcal{C}$ of QLL-formulas. This is a fairly specific

[3] $\to_{\beta c}$ consists of a standard β-reduction relation for λ_c augmented by the reduction $\mathsf{let}\ z \Leftarrow \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ q\ \mathsf{in}\ r \to_c \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow q\ \mathsf{in}\ r$.

true.η	z	$=$	$*$	$(z : \mathit{true})$
false.β	$\mathsf{efq}(y)$	$=$	$\mathsf{efq}(z)$	$(y, z : \mathit{false})$
$\wedge.\beta.1$	$\pi_1(y, z)$	$=$	y	
$\wedge.\beta.2$	$\pi_2(y, z)$	$=$	z	
$\wedge.\eta$	$(\pi_1 z, \pi_2 z)$	$=$	z	$(z : M \wedge N)$
$\vee.\beta.1$	$\mathsf{case}\ \iota_1(y)\ \mathsf{of}\ [\iota_1(z_1) \to q,\ \iota_2(z_2) \to r]$	$=$	$q\{y/z_1\}$	
$\vee.\beta.2$	$\mathsf{case}\ \iota_2(y)\ \mathsf{of}\ [\iota_1(z_1) \to q,\ \iota_2(z_2) \to r]$	$=$	$r\{y/z_2\}$	
$\vee.\eta$	$\mathsf{case}\ y\ \mathsf{of}\ [\iota_1(z_1) \to p\{\iota_1(z_1)/y\},\ \iota_2(z_2) \to p\{\iota_2(z_2)/y\}]$	$=$	p	
$\supset.\beta$	$(\lambda z\,.\,p)\ y$	$=$	$p\{y/z\}$	
$\supset.\eta$	$\lambda z\,.\,(y\ z)$	$=$	y	
$\bigcirc$.ass	$\mathsf{let}\ z \Leftarrow \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ q\ \mathsf{in}\ r$	$=$	$\mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow q\ \mathsf{in}\ r$	
$\bigcirc.\beta$	$\mathsf{let}\ z \Leftarrow \mathsf{val}(y)\ \mathsf{in}\ p$	$=$	$p\{y/z\}$	
$\bigcirc.\eta$	$\mathsf{let}\ z \Leftarrow y\ \mathsf{in}\ \mathsf{val}(z)$	$=$	y	
$\forall.\beta$	$\pi_t\ (\langle p \mid x \rangle)$	$=$	$p\{t/x\}$	
$\forall.\eta$	$\langle (\pi_x\ z) \mid x \rangle$	$=$	z	
$\exists.\beta$	$\mathsf{case}\ \iota_t(p)\ \mathsf{of}\ [\iota_x(z) \to q]$	$=$	$q\{p/z\}\{t/x\}$	
$\exists.\eta$	$\mathsf{case}\ y\ \mathsf{of}\ [\iota_x(z) \to p\{\iota_x(z)/y\}]$	$=$	p	

Figure 4: Equations for $\lambda_c^{\Sigma\Pi}$.

semantics but adequate to capture the standard CLP paradigm. A larger variety of standard and nonstandard constraint semantics may be covered within the general (categorical) framework for $\lambda_c^{\Sigma\Pi}$ axiomatised by the equations in Fig. 4. A class of such more general models, generalising $\lambda_c(\mathcal{C})$, will be introduced in Section 4.2.

4.1 $\mathcal{C}$-Calculi: A Special Class of $\lambda_c^{\Sigma\Pi}$ Models

The class of $\lambda_c(\mathcal{C})$ calculi is generated from a notion of constraint $\mathcal{C}$. They correspond to the "constraints-as-built-in-predicates" paradigm and thus provide for a direct application of the general theory to constraint logic programming CLP. We use the following definition:

Definition 4.2 *A* notion of constraint *for* QLL *is a pair* $\mathcal{C} = (\Phi, \sim)$ *where* Φ *is some class of* QLL *formulas and* $\sim$ *an equivalence relation on* Φ*. These have to satisfy the following conditions: (i)* Φ *contains equality* $x = y$ *and is closed under conjunction* $\wedge$*, existential quantification* $\exists$*, and substitution* $\{t/x\}$*; (ii)* $\sim$ *contains all standard logical equivalences involving* $\wedge$*,* $\exists$*,* $=$*, and in particular is a congruence w.r.t.* $\wedge$*,* $\exists$*; (iii)* $\sim$ *is decidable.*

This definition complies with the standard view adopted in CLP systems [JM94] according to which constraints are a collection of privileged first-order predicates, which are closed under some useful operations, and, most importantly, whose semantics is built in (see *e.g.* [SA89, DGW91, JM94]) so that their computation is precompiled. This precompiled and built-in constraint semantics is captured, in our definition, by the equivalence relation $\sim$. Based on this equivalence we have control over the *extensional* aspects of constraints. For instance, to check the solvability of a constraint c over a single variable x we test whether $(\exists x.\, x = x) \sim (\exists x.\, c)$ holds or not. This is always possible as $\sim$ is required to be decidable. But this is our only concession to implementation issues, here. We disregard the intensional aspect of how $\sim$ is decided, practically. This, of course, depends very much on the particular class of constraints with different classes admitting different algorithms. In some cases, $\sim$ may coincide with provable equivalence in QLL, possibly even provable equivalence in some conveniently implementable fragment of QLL. In other cases $\sim$ may be implemented by dedicated external algorithms, like graph-theoretic analyses to solve, say, systems of one-sided inequations over integers of the form $s \leq t + 106$. In general, when Φ is an undecidable class of predicates, such as nonlinear constraints on real numbers, $\sim$ will only capture some decidable abstraction of the properties expressible by the constraints in Φ. We are free to adjust the parameter $\sim$ and, just as is done in standard CLP systems, compute in this way the semantics of constraints only so far as can be implemented efficiently.

Definition 4.3 *Let $\mathcal{C} = (\Phi, \sim)$ be a notion of constraint for QLL. The calculus $\lambda_c(\mathcal{C})$, called* $\mathcal{C}$-calculus, *is the simply-typed lambda calculus*

- *with finite products ($\times$, $\mathbf{1}$), finite sums ($+$, $\mathbf{0}$), exponentiation ($\Rightarrow$), and two distinguished types $\mathbf{U}$, $\mathbf{C}$,*
- *generated from the well-formed terms of QLL as terms of type $\mathbf{U}$, the constraints $c \in \mathcal{C}$ as terms of type $\mathbf{C}$, and families of operations $\mathsf{val}(\cdot)$, $\mathsf{let}\ z \Leftarrow (\cdot)\ \mathsf{in}\ (\cdot)$ of type $\sigma \Rightarrow \mathbf{C}$,*
- *subject to the typing rules of Fig. 6 and the equations of Fig. 7.*

$$\tau \ ::= \ \mathbf{1} \mid \mathbf{0} \mid \mathbf{U} \mid \mathbf{C} \mid \tau + \tau \mid \tau \times \tau \mid \tau \Rightarrow \tau$$

$$\begin{aligned} p \ ::= \ & x \mid t \mid c \mid \exists x.p \mid p \wedge p \mid * \mid \mathsf{efq}(p) \mid (p,p) \mid \pi_1(p) \mid \pi_2(p) \mid \\ & \mathsf{case}\ p\ \mathsf{of}\ [\iota_1(z_1) \to p,\ \iota_2(z_2) \to p] \mid \iota_1(p) \mid \iota_2(p) \mid \lambda z\,.\,p \mid p\,p \\ & \mathsf{val}(p) \mid \mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ q \end{aligned}$$

x ranges over variables, $t \in \mathcal{U}$, $c \in \mathcal{C}$

Figure 5: Syntax of $\lambda_c(\mathcal{C})$.

$\lambda_c(\mathcal{C})$ can be seen as a concrete model of Moggi's computational lambda calculus λ_c with strong monad $T(\sigma) = \sigma \Rightarrow \mathbf{C}$ for a type $\mathbf{C}$ of constraint propositions. The types τ and terms p of $\lambda_c(\mathcal{C})$ are generated by the abstract syntax shown in Fig. 5. The typing rules and equations for the simply-typed lambda calculus are standard, and the typing and equations for the operations $\mathsf{val}(\cdot)$, $\mathsf{let}\ z \Leftarrow (\cdot)\ \mathsf{in}\ (\cdot)$ are obtained from those of λ_c [Mog91]. Thus, only the remaining typing rules in Fig. 6 and equations in Fig. 7 which are the constraint-specific part of $\lambda_c(\mathcal{C})$, deserve some comments. By assumption every object-level term of QLL is included in $\lambda_c(\mathcal{C})$ as a term of type $\mathbf{U}$, and every object-level variable of QLL is a $\lambda_c(\mathcal{C})$-variable of type $\mathbf{U}$. Consequently, every object-level term t with a free variable x is an open $\lambda_c(\mathcal{C})$-term of type $\mathbf{U}$ with free variable x of type $\mathbf{U}$. The variable can be substituted for by any $\lambda_c(\mathcal{C})$-term s of type $\mathbf{U}$, which is achieved by the typing rule

$$\frac{\Delta \vdash s : \mathbf{U}}{\Delta \vdash t\{s/x\} : \mathbf{U}}\ t \in \mathcal{U}\ ,\ FV(t) \subseteq \Delta, x$$

in Fig. 6, where here and in the following $FV(t)$ is the set of free variables of t. One may wonder, why in a natural deduction style typing system substitution must be accounted for explicitly. The answer is that the more "natural," and simpler rule

$$\frac{}{\Delta \vdash t : \mathbf{U}}\ t \in \mathcal{U}\ ,\ FV(t) \subseteq \Delta, x$$

does not suffice. For then we would not be able to form useful terms of type $\mathbf{U}$, such as $f((\lambda x.s)t)$ or $f(\pi_1(s,t))$ where f is a function symbol of QLL, and s,t object-level terms, which we would like to have available as abstract deconstructions of the concrete object-level terms $f(s\{t/x\})$ or $f(s)$, respectively. A similar argument suggests the typing rule

$$\frac{\Delta \vdash s : \mathbf{U}}{\Delta \vdash c\{s/x\} : \mathbf{C}} \; c \in \Phi_{\mathcal{C}}, FV(c) \subseteq \Delta, x$$

for constraints.

Simply-typed Lambda Calculus

$$\frac{}{\Delta, x : \tau, \Delta' \vdash x : \tau} \qquad \frac{}{\Delta \vdash * : \mathbf{1}} \qquad \frac{\Delta \vdash p : \mathbf{0}}{\Delta \vdash \mathsf{efq}(p) : \tau}$$

$$\frac{\Delta \vdash p : \sigma \quad \Delta \vdash q : \tau}{\Delta \vdash (p,q) : \sigma \times \tau} \qquad \frac{\Delta \vdash r : \sigma \times \tau}{\Delta \vdash \pi_1(r) : \sigma} \qquad \frac{\Delta \vdash r : \sigma \times \tau}{\Delta \vdash \pi_2(r) : \tau}$$

$$\frac{\Delta \vdash r : \sigma + \tau \quad \Delta, y : \sigma \vdash p : \rho \quad \Delta, z : \tau \vdash q : \rho}{\Delta \vdash \mathsf{case}\; r \;\mathsf{of}\; [\iota_1(y) \to p,\; \iota_2(z) \to q] : \rho}$$

$$\frac{\Delta \vdash p : \sigma}{\Delta \vdash \iota_1(p) : \sigma + \tau} \qquad \frac{\Delta \vdash p : \tau}{\Delta \vdash \iota_2(p) : \sigma + \tau}$$

$$\frac{\Delta, z : \sigma \vdash p : \tau}{\Delta \vdash \lambda z \,.\, p : \sigma \Rightarrow \tau} \qquad \frac{\Delta \vdash p : \sigma \Rightarrow \tau \quad \Delta \vdash q : \sigma}{\Delta \vdash p\, q : \tau}$$

Constraints

$$\frac{\Delta \vdash p : \sigma \Rightarrow \mathbf{C} \quad \Delta, z : \sigma \vdash q : \tau \Rightarrow \mathbf{C}}{\Delta \vdash \mathsf{let}\; z \Leftarrow p \;\mathsf{in}\; q : \tau \Rightarrow \mathbf{C}} \qquad \frac{\Delta \vdash p : \sigma}{\Delta \vdash \mathsf{val}(p) : \sigma \Rightarrow \mathbf{C}}$$

$$\frac{\Delta \vdash s : \mathbf{U}}{\Delta \vdash t\{s/x\} : \mathbf{U}} \; t \in \mathcal{U} \,,\; FV(t) \subseteq \Delta, x \qquad \frac{\Delta \vdash s : \mathbf{U}}{\Delta \vdash c\{s/x\} : \mathbf{C}} \; c \in \Phi_{\mathcal{C}}, FV(c) \subseteq \Delta, x$$

$$\frac{\Delta \vdash p : \mathbf{C} \quad \Delta \vdash q : \mathbf{C}}{\Delta \vdash p \wedge q : \mathbf{C}} \qquad \frac{\Delta, x : \mathbf{U} \vdash p : \mathbf{C}}{\Delta \vdash \exists x.\, p : \mathbf{C}}$$

Figure 6: Typing rules of $\lambda_c(\mathcal{C})$.

Let us turn now to the equations of $\lambda_c(\mathcal{C})$. The equations $\mathbf{C}.\mathsf{val}$, $\mathbf{C}.\mathsf{let}$, and $\mathbf{C}.\!\sim$ are the interface between the computational lambda calculus λ_c and the constraints of $\mathcal{C}$. Equation $\mathbf{C}.\!\sim$ turns semantic equivalence of constraints into equality on constraint terms. Given the equational theory thus induced on $\mathbf{C}$ terms is to be conservative, then for it to be "computational," *i.e.* (efficiently) decidable the equivalence $\sim$ had better be (efficiently) decidable in the first place. This is why we require decidability of $\sim$. The other two equations $\mathbf{C}.\mathsf{val}$ and $\mathbf{C}.\mathsf{let}$ (see Fig. 7) allow us to translate the val and let constructs into logic operations on constraints. This is possible in case that the types involved are essentially first-order, which is expressed by the side conditions $x, y, z : \mathbf{U}^k$, where $\mathbf{U}^k$ abbreviates the k-fold product $\mathbf{U} \times \cdots \times \mathbf{U}$. It is possible to relax this restriction to arbitrary first-order types, *i.e.* those obtained from $\mathbf{U}$, $\mathbf{1}$, $\mathbf{0}$, by means of $\times$ and $+$, though the equations would be more complicated then. For our purposes the types $\mathbf{U}^k$ suffice. If, instead of QLL, constraints were taken from a higher-order logic we could have equations $\mathbf{C}.\mathsf{val}$,

$\mathbf{C}.\mathsf{let}$ for arbitrary types. Specifically, $\mathbf{C}.\mathsf{val}$ would become $\mathsf{val}(x)\, y = (x = y)$ and $\mathbf{C}.\mathsf{let}$ would be $(\mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ q)\, x = \exists z.\, p\, z \wedge q\, x$ (see also Example 4.11 in the next section).

We note, as before with $\lambda_c^{\Sigma\Pi}$, that implicit α-conversion and appropriate congruence rules for equality are assumed. This time, the congruence property also must apply to constraint conjunction $\wedge$ and the variable binding operator $\exists$ (ξ-rule). Also, we adopt the usual notions of *free variables*, *open* and *closed* terms. We define a term p to be *quasi-closed* if all its free variables are of type $\mathbf{U}$.

Simply-typed Lambda Calculus

$$
\begin{array}{llcll}
\mathbf{1}.\eta & z & = & * & (z : \mathbf{1}) \\
\mathbf{0}.\beta & \mathsf{efq}(y) & = & \mathsf{efq}(z)(y, z : \mathbf{0}) & \\
\times.\beta.1 & \pi_1(y, z) & = & y & \\
\times.\beta.2 & \pi_2(y, z) & = & z & \\
\times.\eta & (\pi_1 z, \pi_2 z) & = & z \quad (z : \sigma \times \tau) & \\
+.\beta.1 & \mathsf{case}\ \iota_1(y)\ \mathsf{of}\ [\iota_1(z_1) \to q,\ \iota_2(z_2) \to r] & = & q\{y/z_1\} & \\
+.\beta.2 & \mathsf{case}\ \iota_2(y)\ \mathsf{of}\ [\iota_1(z_1) \to q,\ \iota_2(z_2) \to r] & = & r\{y/z_2\} & \\
+.\eta & \mathsf{case}\ y\ \mathsf{of}\ [\iota_1(z_1) \to p\{\iota_1(z_1)/y\},\ \iota_2(z_2) \to p\{\iota_2(z_2)/y\}] & = & p & \\
\Rightarrow.\beta & (\lambda z\,.\, p)\, y & = & p\{y/z\} & \\
\Rightarrow.\eta & \lambda z\,.\,(y\ z) & = & y &
\end{array}
$$

Constraints

$$
\begin{array}{llcll}
\bigcirc.\mathsf{ass} & \mathsf{let}\ z \Leftarrow \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ q\ \mathsf{in}\ r & = & \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow q\ \mathsf{in}\ r & \\
\bigcirc.\beta & \mathsf{let}\ z \Leftarrow \mathsf{val}(y)\ \mathsf{in}\ p & = & p\{y/z\} & \\
\bigcirc.\eta & \mathsf{let}\ z \Leftarrow y\ \mathsf{in}\ \mathsf{val}(z) & = & y & \\
\mathbf{C}.\mathsf{val} & \mathsf{val}(x)\, y & = & \pi_1 x = \pi_1 y \wedge \cdots \wedge \pi_k x = \pi_k y & (x, y : \mathbf{U}^k) \\
\mathbf{C}.\mathsf{let} & (\mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ q)\, x & = & \exists z_1, \ldots, z_k.\, p\,(z_1, \ldots, z_k) \wedge q\, x\{(z_1, \ldots, z_k)/z\} & (z : \mathbf{U}^k) \\
\mathbf{C}.\!\sim & c & = & d & (c, d \in \Phi, c \sim d)
\end{array}
$$

Figure 7: Equations for $\lambda_c(\mathcal{C})$.

We now obtain a specific computational interpretation of $\lambda_c^{\Sigma\Pi}$ by translation into a given $\mathcal{C}$-calculus in the following way:

Definition 4.4 *Let $\mathcal{C}$ be a notion of constraint for QLL. For every assignment of $\lambda_c(\mathcal{C})$ types $|A|$ to atomic formulas A of QLL we define the following translation of $\lambda_c^{\Sigma\Pi}$ into $\lambda_c(\mathcal{C})$:*

1. *Formulas M of QLL are translated into types $|M|$ of $\lambda_c(\mathcal{C})$:*

$$
\begin{array}{rcl}
|true| & := & \mathbf{1} \\
|false| & := & \mathbf{0} \\
|\bigcirc M| & := & |M| \Rightarrow \mathbf{C} \\
|M_1 \wedge M_2| & := & |M_1| \times |M_2| \\
|M_1 \vee M_2| & := & |M_1| + |M_2| \\
|M_1 \supset M_2| & := & |M_1| \Rightarrow |M_2| \\
|\forall x.M| & := & \mathbf{U} \Rightarrow |M| \\
|\exists x.M| & := & \mathbf{U} \times |M|
\end{array}
$$

2. QLL-ND *proofs* p *of a formula* M *are translated into terms* $|p|$ *of* $\lambda_c(\mathcal{C})$*:*

   (a) $|\langle p \mid x\rangle| := \lambda x.|p|$;

   (b) $|\pi_t\, p| := p\, t$;

   (c) $|\mathsf{case}\ r\ \mathsf{of}\ [\iota_x(z) \to p]| := |p|\{\pi_1|r|/x\}\{\pi_2|r|/z\}$;

   (d) $|\iota_t(p)| := (t, |p|)$;

   (e) *All other constructs of* $\lambda_c^{\Sigma\Pi}$ *are translated to themselves;*

Lemma 4.5

(i) For every proof $p : M$ *of* $\lambda_c^{\Sigma\Pi}$*, the translation* $|p|$ *is a well-formed term of* $\lambda_c(\mathcal{C})$ *of type* $|M|$*.*

(ii) The translation respects substitution, i.e. $|p\{q/z\}| = |p|\{|q|/z\}$ *and* $|p\{t/x\}| = |p|\{t/x\}$*.*

Theorem 4.6 *Every assignment of* $\lambda_c(\mathcal{C})$*-types* $|A|$ *to atomic formulas induces a model of* $\lambda_c^{\Sigma\Pi}$*, i.e. a semantics-preserving translation of* $\lambda_c^{\Sigma\Pi}$ *into* $\lambda_c(\mathcal{C})$*.*

The computational semantics of $\lambda_c^{\Sigma\Pi}$ induced by the pair $(|\cdot|, \lambda_c(\mathcal{C}))$ is parametric in the choice of constraint propositions $\Phi_{\mathcal{C}}$, the equivalence $\sim_{\mathcal{C}}$, and an assignment of types $|A|$ of $\lambda_c(\mathcal{C})$ to atomic QLL propositions A. We will consider the case $|R(t_1, \ldots, t_m)| = \mathbf{U}^k$, where $k = \gamma(R) \in \mathbb{N}$ is a function of the relation symbol R.

Although we generally will be interested in the semantics obtained by translating $\lambda_c^{\Sigma\Pi}$ into $\lambda_c(\mathcal{C})$ via $|\cdot|$, it is useful to distinguish between these two levels of λ-calculus systems. The $\lambda_c^{\Sigma\Pi}$ system is used as a method of recording proofs, independent of any particular notion of constraint, whilst the $\lambda_c(\mathcal{C})$ system will be used to capture concrete constraints from CLP program clauses. By reducing the $\lambda_c^{\Sigma\Pi}$ proof term, we calculate the constraint-independent semantics (by the equations of Fig. 4), by reducing its $|\cdot|$-translation in $\lambda_c(\mathcal{C})$ (by the equations of Fig. 7) we evaluate concrete constraints in $\mathcal{C}$. It is usually more advantageous to stay at the more abstract $\lambda_c^{\Sigma\Pi}$ level than to pass to the concrete $\lambda_c(\mathcal{C})$ level too early. We will give an example of this later (Example 6.28). The reductions that will be applied are the $\to_{\beta c\gamma}$-reductions shown in Fig. 8. As usual, a $\lambda_c(\mathcal{C})$-term that cannot be reduced further by $\to_{\beta c\gamma}$ is called a *normal form*.

$$
\begin{array}{rcl}
(\lambda x.p)q & \to_\beta & p\{q/x\} \\
\mathsf{let}\ z \Leftarrow \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ q\ \mathsf{in}\ r & \to_c & \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow q\ \mathsf{in}\ r \\
\mathsf{let}\ z \Leftarrow \mathsf{val}(y)\ \mathsf{in}\ p & \to_c & p\{y/z\} \\
\mathsf{val}(p)\, q & \to_\gamma & \pi_1 p = \pi_1 q \wedge \cdots \wedge \pi_k p = \pi_k q \quad (p, q : \mathbf{U}^k) \\
(\mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ q)\, r & \to_\gamma & \exists z_1, \ldots, z_k.\, p\,(z_1, \ldots, z_k) \wedge q\, r\, \{(z_1, \ldots, z_k)/z\} \quad (z : \mathbf{U}^k)
\end{array}
$$

Figure 8: $\to_{\beta c\gamma}$-reductions for $\lambda_c(\mathcal{C})$.

Proposition 4.7 *The type* $\mathbf{C}$ *of* $\lambda_c(\mathcal{C})$ *is a conservative extension of* $\mathcal{C} = (\Phi, \sim)$*, i.e. (i) for every quasi-closed term* t *of type* $\mathbf{C}$ *there exists some* $c \in \Phi$ *such that* $t \to_{\beta c\gamma} c$*, and (ii) if* $s, t \in \Phi$ *and* $s = t$ *holds in* $\lambda_c(\mathcal{C})$*, then* $s \sim t$*.*

The intensional aspect of actual *constraint solving* is not accounted for by the reductions of Fig. 8. Constraint solving can be viewed as a method of computing normal form representatives of elements $c \in \Phi$ (the $\rightarrow_{\beta c \gamma}$-normal forms of type $\mathbf{C}$) for the equivalence classes modulo $\sim$, see *e.g.* [SA89]. This can be done by reducing constraints into so-called *solved form* or *canonical form*. In our framework this additional "Φ-reduction" would correspond to, for terms $c \in \Phi$, what β-reduction is for the (ambient) lambda calculus. However, there is an important difference in the operational model as compared to lambda calculus in that Φ-reduction may be nondeterministic, *i.e.* not confluent. When computing a solved form to test for solvability efficiently we may want to give up constraint information and merely work with sufficient conditions suggested by heuristic decisions. In such a case the reduction would leave the equivalence class of a constraint. To cover these situations as well, one would replace the equivalence $\sim$ by a partial ordering $\sqsubseteq$ on constraints which captures *relative strength*. We will not explore this here, though.

4.2 Constraint Relations: A General Class of $\lambda_c^{\Sigma\Pi}$ Models

Here we introduce a new calculus Λ_c of *constraint relations* the models of which provide for a very general class of interpretations of $\lambda_c^{\Sigma\Pi}$. The concrete constraint calculus $\lambda_c(\mathcal{C})$ defined in the previous section is but a special case of such a model. The reader may skip this section on first reading since the results presented in the later sections do not depend on it.

Definition 4.8 *A calculus of* constraint relations *for* QLL *is a simply-typed lambda calculus* Λ *with finite products* $(\times,\ \mathbf{1})$*, finite sums* $(+,\ \mathbf{0})$*, exponentiation* $(\Rightarrow)$*, and containing the terms of* QLL *as a sub-language of type* $\mathbf{U}$*; moreover, there is a distinguished type* $\mathbf{C}$ *together with two families of constants*

$$
\begin{array}{rcl}
\eta_\tau & : & \tau \Rightarrow \tau \Rightarrow \mathbf{C} \\
;_{\rho,\sigma,\tau} & : & ((\rho \Rightarrow \sigma \Rightarrow \mathbf{C}) \times (\sigma \Rightarrow \tau \Rightarrow \mathbf{C})) \Rightarrow (\rho \Rightarrow \tau \Rightarrow \mathbf{C})
\end{array}
$$

with ρ, σ, τ *ranging over all types of* Λ*, which satisfy the following equations*

$$
\begin{array}{rcll}
p \;;_{\rho,\sigma,\sigma}\; \eta_\sigma & = & p & (1) \\
\eta_\rho \;;_{\rho,\rho,\sigma}\; p & = & p & (2) \\
p \;;_{\rho,\sigma,\theta}\; (q \;;_{\sigma,\tau,\theta}\; r) & = & (p \;;_{\rho,\sigma,\tau}\; q) \;;_{\rho,\tau,\theta}\; r & (3) \\
p \;;_{\rho,\sigma,\tau}\; q & = & (id_{\sigma\Rightarrow\mathbf{C}} \;;_{\sigma\Rightarrow\mathbf{C},\sigma,\tau}\; q) \circ p, & (4)
\end{array}
$$

where p, q, r *have types* $\rho \Rightarrow \sigma \Rightarrow \mathbf{C}, \sigma \Rightarrow \tau \Rightarrow \mathbf{C}, \tau \Rightarrow \theta \Rightarrow \mathbf{C}$*, respectively. For convenience, the operator* ; *is applied in infix notation;* $\circ$ *is the function composition combinator, i.e.* $f \circ g = \lambda x. f\,(g\,x)$*; id is the identity function,* $id = \lambda x.x$*. Let the free calculus of constraint relations for* QLL *be denoted by* Λ_c*.*

Remark: By assumption every object level term t of QLL is included in Λ as a term of type $\mathbf{U}$, and every object level variable of QLL is a Λ-variable of type $\mathbf{U}$. Thus, every object-level term $t(x_1, \ldots, x_n)$ with free variables x_i is an open Λ-term of type $\mathbf{U}$ with free variables x_i of type $\mathbf{U}$.

Every calculus of constraint relations, or a model of Λ_c, defines the computational part of a *relational notion of constraint*. Our idea is that the objects of type $\tau \Rightarrow \mathbf{C}$ represent a *constraint* on objects of type τ. We may call the type $\mathbf{C}$ the *constraint classifier* of Λ. Given $t : \tau$ and a constraint $c : \tau \Rightarrow \mathbf{C}$, then, intuitively, $c\,t : \mathbf{C}$ represents all information concerning the *solvability* of constraint c for object t that is considered relevant in the given Λ model. A term of type $\sigma \Rightarrow \tau \Rightarrow \mathbf{C}$, *i.e.* a constraint on τ that depends on σ, may be thought of as a constraint on

$\sigma \times \tau$. We can call this a *constraint relation.* In general, a term of type $\tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \mathbf{C}$ is an n-ary constraint relation; A black box with n *ports* of types $\tau_1, \ldots, \tau_n$ which imposes some constraint on a hypothetical environment that connects with these ports. The two families η and ; that come along with every Λ correspond to two kinds of operations on such constraint relations: $\eta_\tau : \tau \times \tau \Rightarrow \mathbf{C}$ is the *identity* relation and $;_{\rho,\sigma,\tau}$ is the *composition* of constraint relations. The first three equations given in Def. 4.8 essentially postulate a monoid of constraint relations with η as neutral element and ; as composition. Using these operations, as well as λ-abstraction and variables of Λ to rearrange ports, we can build up from primitive constraint relations complex *constraint networks*. In fact, the free Λ_c may be seen as a calculus for higher-order constraint networks, such as those of [TU93].

Remark: The type $\mathbf{C}$ represents an *internal category* in Λ_c: The objects of this internal category are the types of Λ_c; the arrows $a : \sigma \rightarrow_{\mathbf{C}} \tau$ are the terms a of type $\sigma \times \tau \Rightarrow \mathbf{C}$, η is the identity arrow, and $p \,;_{\rho,\sigma,\tau}\, q : \rho \rightarrow_{\mathbf{C}} \tau$ the composition of arrows $p : \rho \rightarrow_{\mathbf{C}} \sigma$ and $q : \sigma \rightarrow_{\mathbf{C}} \rho$. The equations (1), (2), (3) in Def. 4.8 are precisely the monoid laws for this internal category. The remaining equation (4) ensures that the two composition operations $\circ$ in Λ_c and ; in $\mathbf{C}$ are coherent, that "whenever possible" they can be expressed in terms of each other. Thus, in category-theoretic terms a constraint calculus is a bicartesian[4] closed category $\mathbb{C}$ with a type $\mathbf{U}$ of QLL terms and an internal category object $\mathbf{C}$.

Remark: There is an equivalent category-theoretic characterisation in terms of monads, which is perhaps more in the spirit of Moggi's original conception of computational lambda calculi. A constraint calculus is a bicartesian closed category with an object $\mathbf{C}$ such that the map $\tau \mapsto \tau \Rightarrow \mathbf{C}$ is a strong monad. One can show, using the equations (1)–(4), that the mapping F translating a function $f : \sigma \Rightarrow \tau$ of into the arrow $F\,f = \eta_\tau \circ f : \sigma \rightarrow_{\mathbf{C}} \tau$ is a functor from Λ_c to the internal category $\mathbf{C}$. Also, the mapping G which takes an object τ of $\mathbf{C}$ to the type $\tau \Rightarrow \mathbf{C}$ of Λ_c, and an arrow $g : \sigma \rightarrow_{\mathbf{C}} \tau$ of $\mathbf{C}$ to the function $G\,g = id \,;\, g : (\sigma \Rightarrow \mathbf{C}) \Rightarrow (\tau \Rightarrow \mathbf{C})$ is a functor from the internal category to Λ_c. It turns out that G the right adjoint to F, with η as unit and id as counit. Moreover, this adjunction is so that the functor GF from Λ_c to Λ_c is a strong monad.

Example 4.9 *The standard set-theoretic model of the pure simply-typed lambda calculus can be extended to a model of Λ_c by taking $\mathbf{C}$ to denote some fixed complete lattice. The identity η on τ is defined so that for elements $x, y \in \tau$, $\eta\,(x, y)$ is the top element $\top$ of the lattice if x and y are identical and the bottom element $\perp$ otherwise. Composition $q \circ p$ is defined as $(q \circ p)(r, t) = \bigvee \{\, p(r, s) \wedge q(s, t) \mid s \in \sigma \,\}$, where $\wedge$ is the meet and $\bigvee$ the supremum in $\mathbf{C}$.*
A distinguished such "lattice" model is where $\mathbf{C}$ is the set $\mathbb{B} = \{0, 1\}$ of Booleans with $0 \leq 1$. The information recorded by elements of $\mathbf{C} = \mathbb{B}$ is whether or not the respective constraint holds. A constraint $c : \tau \Rightarrow \mathbb{B}$ on τ is simply a subset of τ, a constraint relation $p : \rho \Rightarrow \sigma \Rightarrow \mathbb{B}$ a set-theoretic relation. η is the identity relation and ; is relation composition.

Example 4.10 *The trivial two-element model of Λ_c consists of a distinguished singleton set $1 = \{*\}$ and the empty set $\mathbf{0} = \emptyset$. The types $\mathbf{U}$ and $\mathbf{C}$ are identified with $\mathbf{1}$. This model corresponds to the proof collaps model in which no constraint information is recorded and we are interested only in the extensional fact that a formula of* QLL *is true.*

The next examples link up the notion of constraint relation with the calculus $\lambda_c(\mathcal{C})$, essentially showing that $\lambda_c(\mathcal{C})$ is a special version of a calculus of constraint relations.

[4] a category that is finitely complete, finitely cocomplete, and has exponentials.

Example 4.11 *Let $\mathcal{C}$ be a distinguished subclass of formulas of some many-sorted higher-order logic with term language $\mathcal{U}$ and equality $=$. We construct a special theory $\Lambda_c(\mathcal{C})$, and hence a model, of Λ_c as follows: We consider every formula $M \in \mathcal{C}$ as a new term of type* **C**, *and identify the elements of* **C** *up to provable logic equivalence. Then, if M has free (object) variables $x_1, \ldots, x_n$ we have a $\Lambda_c(\mathcal{C})$ term $\lambda x_1. \cdots \lambda x_n. M$ of type $\mathbf{U} \Rightarrow \cdots \Rightarrow \mathbf{U} \Rightarrow \mathbf{C}$. Then, we interpret the identity constraint $\eta_{\mathbf{U}} : \mathbf{U} \times \mathbf{U} \Rightarrow \mathbf{C}$ over the universe type* **U** *with equality $\lambda x.\lambda y.x = y$, and the composition $;_{\mathbf{U},\mathbf{U},\mathbf{U}}$ by a combination of conjunction and existential quantification $\lambda z.\lambda x.\lambda y.\, \exists v.\, (\pi_1 z)\, x\, v \wedge (\pi_2 z)\, v\, y$. Note that by identifying constraint formulas only up to provable equivalence rather than semantic equivalence and the fact that higher-order logic is incomplete,* **C** *essentially becomes a type of non-classical truth-values. This model, which we refer to as $\Lambda_c(\mathcal{C})$, thus is more intensional than the set-theoretic one with* **C** *being the lattice of Boolean truth values (see Example 4.9).*

Example 4.12 *Every topos is a model of Λ_c. The constraint type is the subobject classifier Ω. The operators η and $;$ are defined as in the previous example.*

We now obtain a specific computational interpretation of $\lambda_c^{\Sigma\Pi}$ by translation into a given calculus of constraint relations in the following way:

Definition 4.13 *Let Λ be a calculus of constraint relations. For every assignment of Λ types $|A|$ to atomic formulas A of* QLL *we define the following translation of $\lambda_c^{\Sigma\Pi}$ to Λ:*

1. *Formulas M of* QLL *are translated into types $|M|$ of Λ as in Definition 4.4.*
2. QLL-ND *proofs p of a formula M are translated into terms $|p|$ of Λ of type $|M|$:*
   - *(a)* $|\mathsf{val}(p)| := \eta_{|M|}\, |p|$;
   - *(b)* $|\mathsf{let}\ x \Leftarrow p\ \mathsf{in}\ q| := (id_{|M| \Rightarrow \mathbf{C}}\ ;\ (\lambda x.\, |q|))\, |p|$;
   - *(c) The remaining constructs are translated as in Definition 4.4.*

Lemma 4.14

(i) For every proof $p : M$ of $\lambda_c^{\Sigma\Pi}$, the translation $|p|$ is a well-formed term of Λ of type $|M|$.

(ii) The translation respects substitution, i.e. $|p\{q/z\}| = |p|\{|q|/z\}$ and $|p\{t/x\}| = |p|\{t/x\}$.

Theorem 4.15 *Every calculus Λ of constraint relations is a model of $\lambda_c^{\Sigma\Pi}$, the interpretation given by the translation $|\cdot|$.*

5 Lax Logic Programming LLP

In this section we introduce the LLP-fragment of QLL for Lax Logic programming. LLP acts as a language for both actual, concrete CLP programs and their abstract forms, as introduced in the next section. LLP has a similar relationship to QLL as the fragment of Prolog has to full intuitionistic predicate logic. We also introduce two sub-structural calculi of QLL, $\vdash_{\mathsf{r}}$ and $\vdash_{\mathsf{l}}$, which are two different formalisations of constrained SLD resolution, the standard goal-directed backward operational semantics [JM94] of logic programming.

Definition 5.1 *A* Σ-formula *is a formula generated by the following grammar:*

$$S \quad ::= \quad \mathit{true} \mid A \mid S \wedge S \mid S \vee S \mid \exists x.\, S,$$

where A is an atomic (program or constraint) formula of the language $\mathcal{L}$. A LLP-clause *is a formula θ of the form*

$$\theta \quad = \quad \forall x_1, \ldots, x_m.\, S \supset H,$$

where H is an atom $P(x_1, \ldots, x_n)$ or a modalised atom $\bigcirc P(x_1, \ldots, x_n)$, such that all free variables of S are in the set $\{x_1, \ldots, x_m\}$ and $m \geq n$. H is called the head *of the clause θ. Σ-formulas and LLP-clauses are called* LLP-formulas*. A LLP*-program *is a finite list $\Pi = \theta_1, \ldots, \theta_n$ of LLP-clauses.*

Σ-formulas are useful to express complex queries and to form the Clark *completion* [Cla78] of a (program) predicate, *i.e.* to package up all program clauses for a given predicate into one self-contained clause. In using the term Σ-formula for this class of formulas we follow [AV96]. In the literature Σ-formulas are also called *goal* formulas or *queries*, *e.g.* in [And92]. Here, queries H may also be modalised Σ-formulas $\bigcirc S$.

Two particular types of LLP-clauses will concern us. The first are constraint-free clauses with modalised heads, called *abstract clauses.* The other type of clause may contain constraints, but the head is not modalised and not a constraint. These are called *CLP-clauses.* A LLP-program in which all clauses are CLP-clauses is a *CLP-program* and one in which all clauses are abstract clauses is called an *abstract program.*

The basic idea behind using a logic calculus as an operational semantics for constraint logic programming is this: If Π is a program and H a query, then we use the rules of the calculus to search for a proof $\vec{w} : \Pi \vdash p : H$. From the proof p, then, given some suitable semantic interpretation we obtain an answer constraint. For LLP-programs one such calculus is QLL-ND and a proof semantics is $|\cdot|$. Although much more structure-directed than a Hilbert calculus, the natural deduction calculus of QLL, in a sense, still gives far too loose an operational semantics in that it still leaves a lot of room for proof search strategies. For practical purposes one would like to use more specific calculi, in which proof search is more focused and less blind. Such calculi exists for standard logic programming and they exist for LLP-programs. The first sub-calculus of QLL which we will consider for LLP is the natural deduction system obtained by the derived introduction and elimination rules given in Fig. 9, together with all introduction rules $\mathit{true}_{\mathcal{I}}$, $\wedge_{\mathcal{I}}$, $\vee_{\mathcal{I}}$, and $\exists_{\mathcal{I}}$ found in Fig. 2.
The reader can check that the following may serve as definitions of the derived λ-terms:

$$\begin{aligned}
\mathit{true}_{\bigcirc} &:= \mathsf{val}(*) \\
\wedge_{\bigcirc}(p, q) &:= \mathsf{let}\ y \Leftarrow p\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow q\ \mathsf{in}\ \mathsf{val}(y, z) \\
\vee_{\bigcirc}(p, i) &:= \mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ \mathsf{val}(\iota_i(z)) \qquad (i = 1, 2) \\
\supset_{\bigcirc}(p, w, \vec{t}) &:= \mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ w\, \vec{t}\, z \\
\exists_{\bigcirc}(p, t) &:= \mathsf{let}\ z \Leftarrow p\ \mathsf{in}\ \mathsf{val}(\iota_t(z)) \\
\supset_{\mathcal{N}}(p, w, \vec{t}) &:= w\, \vec{t}\, p.
\end{aligned}$$

The $\vdash_r$ calculus of Fig. 9 is a calculus of *simple uniform proofs*, for formulas of the LLP-fragment, characterised by the requirement that introduction rules have precedence over elimination rules (uniformity), and that the only elimination rule is the *backchaining* rule $\supset_{\mathcal{N}}$ (simple-ness). Such

$$\frac{}{\Pi \vdash_{\mathrm{r}} \mathit{true}_{\bigcirc} : \bigcirc \mathit{true}}\ \mathit{true}_{\bigcirc} \qquad \frac{\Pi \vdash_{\mathrm{r}} p : \bigcirc S\{t/x\}}{\Pi \vdash_{\mathrm{r}} \exists_{\bigcirc}(p,t) : \bigcirc \exists x.\, S}\ \exists_{\bigcirc} \qquad \frac{\Pi \vdash_{\mathrm{r}} p : \bigcirc S \quad \Pi \vdash_{\mathrm{r}} q : \bigcirc T}{\Pi \vdash_{\mathrm{r}} \wedge_{\bigcirc}(p,q) : \bigcirc(S \wedge T)}\ \wedge_{\bigcirc}$$

$$\frac{\Pi \vdash_{\mathrm{r}} p : \bigcirc S}{\Pi \vdash_{\mathrm{r}} \vee_{\bigcirc}(p,1) : \bigcirc(S \vee T)}\ \vee_{\bigcirc} \qquad \frac{\Pi \vdash_{\mathrm{r}} q : \bigcirc T}{\Pi \vdash_{\mathrm{r}} \vee_{\bigcirc}(q,2) : \bigcirc(S \vee T)}\ \vee_{\bigcirc}$$

$$\frac{\Pi \vdash_{\mathrm{r}} p : \bigcirc S\{\vec{t}/\vec{x}\}}{\Pi \vdash_{\mathrm{r}} \supset_{\bigcirc}(p,w,\vec{t}) : \bigcirc A\{\vec{t}/\vec{x}\}}\ \supset_{\bigcirc} \quad \text{(where } w : \forall \vec{x}.\, S \supset \bigcirc A \text{ is a clause of } \Pi)$$

... introduction rules $\mathit{true}_{\mathcal{I}}$, $\wedge_{\mathcal{I}}$, $\vee_{\mathcal{I}}$, and $\exists_{\mathcal{I}}$ (Fig. 2) ...

$$\frac{\Pi \vdash_{\mathrm{r}} p : S\{\vec{t}/\vec{x}\}}{\Pi \vdash_{\mathrm{r}} \supset_{\mathcal{N}}(p,w,\vec{t}) : A\{\vec{t}/\vec{x}\}}\ \supset_{\mathcal{N}} \quad \text{(where } w : \forall \vec{x}.\, S \supset A \text{ is a clause of } \Pi)$$

Figure 9: LLP backward rules for SLD resolution.

systems are tailored towards a goal-directed backward proof search, and enjoy a close correspondence to the SLD resolution scheme of logic programming [DG94]. When used as an operational semantics, the rules of $\vdash_{\mathrm{r}}$ are applied in a backward fashion, so as to build up a proof tree that grows from the root. In order to execute a program Π with query H one starts with the sequent $\vec{w} : \Pi \vdash_{\mathrm{r}} z : H$ as the root and only leaf of a single-node proof tree. The proof variable z is a unification variable which is gradually instantiated in the construction, while variables $\vec{w}$ are not supposed to be unified. Following standard terminology z may be called *flexible* and $\vec{w}$ *rigid* variables. The tree grows by expanding leaves. A leaf is expanded, first by picking a rule whose conclusion sequent unifies with the leaf sequent, and then by adding the instantiated premiss sequent(s) of the rule to the tree, as the leaf's new successor nodes. If the rule was an axiom, *i.e.* without premisses, then the leaf is closed off and marked as *successful*. All other leaves are marked as *dangling*. If a proof tree has been built with root $\vec{w} : \Pi \vdash_{\mathrm{r}} p : H$ in which all leaves are successful, the execution terminates with the proof term p as the answer.

$$\frac{}{\Pi, z : S \vdash_{\mathrm{l}} \mathsf{val}(z) : \bigcirc S} \qquad \frac{}{\Pi, z : S \vdash z : S} \qquad \frac{\Pi, z : \mathit{true} \vdash_{\mathrm{l}} p : H}{\Pi \vdash_{\mathrm{l}} p\{*/z\} : H}$$

$$\frac{\Pi, z : S_1 \wedge S_2 \vdash_{\mathrm{l}} p : H}{\Pi, v_1 : S_1, v_2 : S_2 \vdash_{\mathrm{l}} p\{(v_1,v_2)/z\} : H} \qquad \frac{\Pi, z : \exists x.\, S \vdash_{\mathrm{l}} p : H}{\Pi, v : S\{t/x\} \vdash_{\mathrm{l}} p\{\iota_t v/z\} : H}$$

$$\frac{\Pi, z : S_1 \vee S_2 \vdash_{\mathrm{l}} p : H}{\Pi, v : S_1 \vdash_{\mathrm{l}} p\{\iota_1 v/z\} : H} \qquad \frac{\Pi, z : S_1 \vee S_2 \vdash_{\mathrm{l}} p : H}{\Pi, v : S_2 \vdash_{\mathrm{l}} p\{\iota_2 v/z\} : H}$$

$$\frac{\Pi, z : A\{\vec{t}/\vec{x}\} \vdash_{\mathrm{l}} p : H}{\Pi, v : S\{\vec{t}/\vec{x}\} \vdash_{\mathrm{l}} \mathsf{let}\ z \Leftarrow w\,\vec{t}\,v\ \mathsf{in}\ p : H}\ w : \forall \vec{x}.\, S \supset \bigcirc A \text{ in } \Pi$$

$$\frac{\Pi, z : A\{\vec{t}/\vec{x}\} \vdash_{\mathrm{l}} p : H}{\Pi, v : S\{\vec{t}/\vec{x}\} \vdash_{\mathrm{l}} p\{w\,\vec{t}\,v/z\} : H}\ w : \forall \vec{x}.\, S \supset A \text{ in } \Pi$$

Figure 10: LLP forward rules for SLD resolution.

If we restrict the calculus $\vdash_{\mathrm{r}}$ to CLP-programs and drop all modality rules we get the sequent calculus proposed by [DG94][5] as a formalisation of constrained SLD resolution. A different system for SLD resolution is the sequent calculus $\vdash_{\mathrm{l}}$ given in Fig. 10. It, too, is a sub-structural calculus of QLL, *i.e.* all its rules can be derived from those of QLL. However, in contrast to $\vdash_{\mathrm{r}}$, it is designed to be applied in a *forward* way. As may be worked out from the rules in Fig. 10 the deconstruction of the goal takes place on the left hand side of the sequent turnstile $\vdash_{\mathrm{l}}$. Given a program Π and a query $\bigcirc S$, we start from an initial sequent $\vec{w} : \Pi, z : S \vdash_{\mathrm{l}} \mathsf{val}(z) : \bigcirc S$ and derive a sequence of sequents[6]

$$\rightsquigarrow\ \vec{w} : \Pi, z : S \vdash_{\mathrm{l}} \mathsf{val}(z) : \bigcirc S\ \rightsquigarrow\ \cdots,\ \rightsquigarrow\ \vec{w} : \Pi, v_1 : S_1, \ldots, v_n : S_n \vdash_{\mathrm{l}} q : \bigcirc S,$$

where each $\rightsquigarrow$ is an application of a rule of Fig. 10. Each sequent in this forward derivation can be viewed as a *proof state*, with $\vec{w} : \Pi, z : S \vdash_{\mathrm{l}} \mathsf{val}(z) : \bigcirc S$ being the initial state. The derivation may carry on until it reaches a state $\vec{w} : \Pi \vdash_{\mathrm{l}} p : \bigcirc S$. The proof term p, then, represents the answer constraint for the query S.

Remark: A technical note on variables is in order here. In the forward application of the $\vdash_{\mathrm{l}}$-rules new variables v, v_1, v_2 and $\vec{u}$ must be generated. The choice of these variables is unconstrained except that they must be fresh, *i.e.* not appear already in the sequents.

There are reasons to consider the sequent calculus $\vdash_{\mathrm{l}}$ (under forward reasoning) as a more adequate formalisation of constrained SLD resolution than $\vdash_{\mathrm{r}}$ (under backward reasoning). First of all, proof search in $\vdash_{\mathrm{l}}$ does not need unification and flexible variables. This means that more work is done within the calculus rather than outside at the meta-level.
Secondly, we note that if we cannot reach the final sequent $\vec{w} : \Pi \vdash_{\mathrm{l}} p : \bigcirc S$ or if we stop before, the sequent we end up with is a valid sequent, nevertheless. In other words, proof states in $\vdash_{\mathrm{l}}$ are sequents, whereas in $\vdash_{\mathrm{r}}$ proof states are deduction trees. This is because the information we have obtained in an intermediate $\vdash_{\mathrm{r}}$-proof state resides in the validity of the intermediate tree as a derived rule. We only have a valid sequent if all the leaves are successful. This implies that if proof states are to represent formally the execution states of constrained SLD resolution, these execution states are sequents in $\vdash_{\mathrm{l}}$ while they are trees of sequents in $\vdash_{\mathrm{r}}$. Thus, the formalisation of $\vdash_{\mathrm{l}}$ is more elementary. Concretely, a sequent $\vec{w} : \Pi, v_1 : S_1, \ldots, v_n : S_n \vdash_{\mathrm{l}} p : \bigcirc S$ corresponds to an execution state of constrained SLD resolution with the list $S_1, \ldots, S_n$ being the current list of goals and p containing the current contents of the constraint store.
A third justification of $\vdash_{\mathrm{l}}$ is a technical one: The $\lambda_c(\mathcal{C})$ proof terms generated by $\vdash_{\mathrm{l}}$ are in normal form[7], which they are not in $\vdash_{\mathrm{r}}$. This has the effect that for $\vdash_{\mathrm{l}}$ constraint extraction is more direct.

The relationship between the systems is clarified by the following theorem. It says that as far as provability is concerned both systems $\vdash_{\mathrm{r}}$ and $\vdash_{\mathrm{l}}$ essentially are equivalent.

Theorem 5.2 *Let Π be a LLP-program, $S, S_1, \ldots, S_n$ a list of Σ-formulas, and $H = S$ or $H = \bigcirc S$. Then, $\Pi, S_1, \ldots, S_n \vdash_{\mathrm{l}} H$ iff there exists a partial proof (= proof tree) $\Pi \vdash_{\mathrm{r}} H$ with dangling leaves $\Pi \vdash_{\mathrm{r}} \bigcirc S_i$, $i = 1, \ldots, n$, if H is modalised, and $\Pi \vdash_{\mathrm{r}} S_i$ otherwise.*

It will be convenient to write $\vec{w} : \Pi, z_1 : S_1, \ldots, z_n : S_n \vdash_{\mathrm{r}} p : \bigcirc S$ for an incomplete proof $\vec{w} : \Pi \vdash_{\mathrm{r}} p : \bigcirc S$ with dangling leaves $\vec{w} : \Pi \vdash_{\mathrm{r}} z_i : \bigcirc S_i$, $i = 1, \ldots, n$. This notational abbreviation is justified by Theorem 5.2. Henceforth, we use the term LLP-calculus, or simply LLP, and write $\vdash_{\mathrm{LLP}}$ to refer to one of the two sub-structural calculi $\vdash_{\mathrm{r}}, \vdash_{\mathrm{l}}$ of QLL.

[5] The calculus of [DG94] also includes atomic constraint reasoning which in our case is contained in the semantics of proofs.

[6] If we wish to prove a nonmodal query S we start from the sequent $\vec{w} : \Pi, z : S \vdash_{\mathrm{l}} z : S$.

[7] This is a bit simplified for the sake of the argument. A single application of $\bigcirc.\eta$ contraction must be done.

Of course, there are other sub-structural calculi for LLP conceivable, which are derived or admissible for QLL, formalising different operational semantics for LLP. As long as the translation into QLL is constructive the method of constraint extraction presented in this work will apply. Such calculi may even be of higher-order nature, involve QLL formulas outside of the LLP-fragment, and cut rules. Both systems $\vdash_{\mathrm{l}}$ and $\vdash_{\mathrm{r}}$ presented above are special in that they do not have a cut rule and always stay within the class of LLP-formulas. They can be seen as a compositional, or fine-grained, versions of an operational semantics in which every program construct $\forall, \exists, \wedge, \vee, \supset, \bigcirc$ gets a separate logical meaning. If we take the modality to refer to the standard case of equality constraints on Herbrand terms, then the calculus $\vdash_{\mathrm{r}}$ is related to the "one-formula" operational semantics Csa, and $\vdash_{\mathrm{l}}$ to the "one-stack" operational semantics OS of [And92], and the calculus LC(Π) of [HSH90]. It is important to stress that while $\vdash_{\mathrm{l}}$ is a genuine logic calculus, with independent model-theoretic semantics, the systems OS of [And92], and the calculus LC(P) of [HSH90] do not have a logical status. They are merely operational semantics for programs that happen to be logic formulas. We conjecture that also the other operational semantics of [And92] can be captured[8] by different sub-structural calculi of QLL. Also, LLP-calculi for the standard forward generative model (see *e.g.* [JM94]) are likely to exist.

Thus, QLL has the same rôle of reference for a given LLP-calculus as the calculus $\vdash_{\mathrm{hc}}$ of [DG94] has for the sub-calculus of simple uniform proofs. Having said that, it is interesting to note that QLL, even without any constraint reasoning, is more powerful than $\vdash_{\mathrm{hc}}$ with constraint reasoning. This is because in $\vdash_{\mathrm{hc}}$ constraints are required to be atomic formulas, so that the constraint-forming operations $\exists, \wedge$ cannot be decomposed within $\vdash_{\mathrm{hc}}$. This means, for instance, that we may be able to prove $\Theta, s \geq 5, s \geq 9, t \geq s + 35 \vdash_{\mathrm{hc}} B(t)$ but not $\Theta, \exists s.\, s \geq 5 \wedge s \geq 9 \wedge t \geq s + 35 \vdash_{\mathrm{hc}} B(t)$ for some CLP-program Θ, which however is possible in QLL. Such a composite constraint is extracted from LLP-proofs, as will be seen in Example 6.27. This deficiency of $\vdash_{\mathrm{hc}}$ has the effect that the completeness theorems of [DG94] are not as strong as they could be.

In proposing QLL as a framework to formalise CLP semantics we deviate from other proof-theoretic approaches, such as the sequent calculus of [DG94], in one important respect. LLP is not meant to contain explicit constraint reasoning. From the point of view of LLP constraints are not first-class propositions but intensional refinement information for propositions, whence they must live and be dealt with at another level, *viz.* at the level of proofs. This separation of concerns is in the same spirit as the stratified calculus of [DG94] but formalised in LLP into a different dimension. In LLP we envisage the following two-dimensional picture:

$$
\begin{array}{lcl}
c,\, d \text{ constraints} & \begin{tikzcd} M \arrow[r, bend left=60, "c"] \arrow[r, bend right=60, "d"'] \arrow[r, phantom, "\Downarrow"] & N \end{tikzcd} & \begin{array}{l} \curvearrowright \ \text{abstract proof} \\ \quad = \text{constraint generation} \\[1ex] \Downarrow \ \text{constraint analysis} \\ \quad = \text{proof semantics} \end{array} \\
M, N \text{ formulas} & &
\end{array}
$$

LLP is intended to formalise an abstract view of CLP-programs that is concerned with *constraint generation*, while *constraint reasoning* is a side-effect delegated to the semantics of proofs. We feel that this is a very natural and technically expedient way of organising matters, and a central idea of our framework. The suggested separation can be undone in a sufficiently rich all-encompassing logic if one wishes to do so. We will see that for LLP the *constraint : formula* split can be interpreted in a suitable theory of QLL, specifically in QLL(=).

As indicated, our idea is to use the LLP calculus to simulate the execution of CLP-programs so that the proofs returned represent the answer constraint. To do this the CLP-program typically is

[8] For the "one-stack" sequential-or semantics OS/so of [And92] intuitionistic negation is needed to express finite failure.

first abstracted into an abstract LLP-program so that constraints are out of the way and pushed into the proofs. Without thus abstracting, in general, a query H cannot be proven in LLP from a CLP program Π directly, since every proof attempt runs across explicit constraints for which LLP has no rules. Depending on the particular LLP-calculus different things can happen. In the $\vdash_{\mathrm{l}}$ system we will end up with a sequent $\Pi, B_1, \ldots, B_n \vdash_{\mathrm{l}} H$ where the B_i are constraints. In $\vdash_{\mathrm{r}}$ we get a proof tree with dangling leaves all of which of the form $\Pi \vdash B$ or $\Pi \vdash \bigcirc B$ where B is a constraint. Both cases are called a *partial proof*, written $\Pi, B_1, \ldots, B_n \vdash_{\mathrm{LLP}} H$. Partial proofs can be seen as proofs of $\Pi \vdash_{\mathrm{LLP}} H$ with constraint holes.

6 Abstraction and Refinement

One motivation for this work is the observation that CLP constraints are not just propositions to be handled at the same level as user-programs, with the only exception that they have a predefined model-theoretic semantics. If constraints are propositions, so we argue, then they are of rather special nature and, from a proof-theoretic point of view, deserve a more dedicated technical treatment. In particular, we wish to advance the idea that constraints are closely connected to abstractions, according to the paradigmatic definition

> *A constraint is a condition on the environment of a system under which a particular abstraction of its behaviour is valid*

as it applies to constraints in software and hardware design. In this section we try to show that this definition applies to constraints in CLP programming, too. We show that an answer constraint can be thought of as generated by the execution of an abstraction of a CLP program, so that the answer constraint makes the query a valid consequence of the original concrete CLP program. In fact, if we strip off all constraints from a CLP program retaining only the user-defined predicates, then, disregarding satisfiability checks, the execution of the program completely is controlled by the abstracted, *i.e.* stripped, program. This operational view, which contains the essence of constraint programming, has been formalised in the operational model proposed in [DGW91] which aims at a clean separation between constraint generation and constraint analysis. Thus, from an operational viewpoint it is the abstracted user-program that takes the primary control while constraints arise as a by-product of executing the program, which is quite in line with the above paradigmatic definition.

Now, of course, we must not ignore the issue of constraint satisfiability. In practice, the constraint level does influence the proof search of the abstract level through the check for constraint satisfiability, which would appear to jeopardise the abstraction idea. This is true to a certain extent. However, the influence of constraint predicates on the execution is of a fundamentally different, and weaker, nature than that of the user-predicates. In executing a CLP-program our goal it to *prove* all user-predicates we run across, while we are content with merely checking the *consistency* of the constraints, which is an intrinsically simpler problem than proving a formula. So there is an abstraction here as well. The weaker operational rôle is not just convenience but lies in the nature of constraints. In fact, it hardly makes sense nor is it possible to prove the validity of a constraint. An answer constraint c resulting from running a CLP program Θ with a query Q verifies $\Theta \vdash \forall \vec{x}. c \supset Q$. Here, c formally is a condition on the environment, *viz.* the instantiations of the variables $\vec{x}$, for which the query Q is valid. We are interested in the constraint c as a description of the class of satisfying environments not in proving it valid, for this would require we close off the modelled system by fixing a particular environment. The only sensible thing we can do, facing incomplete knowledge about the environment, is to make sure that the constraint is satisfiable, *i.e.* that there exist some validating environment at all. However, this check may not be feasible until further instantiations have taken place. In practice it is useful to be able to delay the satisfiability test and to explore the abstract search space [Pla81], before one turns to checking the satisfiability of the accumulated constraints. An example is the timing analysis of combinational circuits. There, one typically first computes a rough estimate of the propagation delay by searching through the abstract topological structure of the circuit completely, and then, once this is done, takes into account the data dependencies to eliminate the so-called "false paths" that have generated unsatisfiable functional constraints [MB91]. The topological delay computed in the abstract domain is used to prune the search at the concrete level.

Here we propose a quite general method of decoupling the generation of constraints and their analysis, which exploits the close relationship between the pragmatically motivated dichotomy *constraint : query* of CLP and the logic dichotomy *proof : formula* of QLL. In this setting, constraint generation is controlled by searching for a proof for an abstraction of the program, and constraint analysis corresponds to a refinement of the abstract query, whose purpose is to fill in the gaps and to map back and justify the abstract proof w.r.t. the concrete level. The answer constraint, then, is nothing but a measure of the extent to which the abstract proof can be justified. This "justification" procedure is implicit in publications applying abstractions to theorem proving, *e.g.* in [GW92, Pla81], even if the connection with constraints in not verbalised. The power of the constraints-as-proofs paradigm lies in the fact that it provides for a uniform and rigorous way of abstracting out all constraints from their local contexts, and of collecting them up into a single and global constraint of the abstracted formula. Consider the formula $\forall x.\, c \supset Q$. If constraints are restricted to be just propositions then it is hard to imagine any reasonable way of abstracting out the constraint c and to make it a constraint of the whole formula, other than in the form $(\forall x.\, c) \supset (\forall x.\, Q)$. But this does not help a lot since the new constraint $(\forall x.\, c)$ is too strong, possibly even inconsistent, in general to be useful. In our framework where constraints are part of the proofs we can abstract out the constraint as $\lambda x.c : \forall x.\, \bigcirc Q$. This pair, in a mathematically precise sense, is a formal *assertion-guarantee* implication "assume $\forall x.\, Q$ in all contexts in which the proof $\lambda x.c$ reduces to a valid constraint." In this way we do not loose any information and yet the constraint is separated from the abstract formula.

Technically, the QLL scenario for abstraction and refinement is as follows: given a CLP program Θ and a query Q, we first abstract out all constraints from Θ to give a pair $\Xi : \Pi$ where Ξ is a *constraint table* and Π the *abstract version* of Θ. Then, the query $\bigcirc Q$ is verified in QLL, obtaining a proof $\Xi : \Pi \vdash p : \bigcirc Q$. By refining the abstract pair $p : \bigcirc Q$ back into a concrete formula we obtain c such that $c \supset Q$ is a logic consequence of Θ. The constraint c is determined by the computational semantics of p, and is composed, in general, from the constraints of the program clauses by the operations $\wedge$, $\exists$, and substitution. By changing the computational semantics of $\lambda_c^{\Sigma\Pi}$ proofs, we can capture different notions of constraint.

To start off our presentation of the technical details we need to make some preparations. From now on we will assume that $\mathcal{C} = (\Phi_{\mathcal{C}}, \sim_{\mathcal{C}})$ is some (notion of) constraint for QLL and that we have assigned to every primitive relation symbol R a number $\gamma(R) \in \mathbb{N}$. For every relation R, then, this $k = \gamma(R)$ indicates the number of implicit constraint parameters of R. To be more specific, if $R(t_1, \dots, t_n)$ is a sub-formula of a program then we consider the first k arguments of R as *constraint parameters* that we wish to distill out from the formula, *viz.* by replacing $R(t_1, \dots, t_n)$ formally by $\mathsf{val}(t_1, \dots, t_k) : \bigcirc R(t_{k+1}, \dots, t_n)$. To do this we must define our translation $|\cdot|$ so that $|R(t_1, \dots, t_n)| = \mathbf{U}^k$. Since constraints will be abstracted not piecewise but wholesale the parameter $\gamma(R)$ has no importance when $R(t_1, \dots, t_n) \in \Phi_{\mathcal{C}}$; we put $\gamma(R) = 0$ in this case, with the effect that the associated proof information has the trivial type $\mathbf{U}^0 = \mathbf{1}$. In our examples we will make further assumptions, in particular about $\mathcal{C}$ and $\gamma(R)$, as appropriate.

The details of this section are complemented by a running example started here. A concrete CLP program is introduced which first undergoes what amounts to *horizontal decomposition* or a *separation of concerns*. That is to say, we separate the CLP program into an abstraction and an associated set of constraints. It is then shown how we can recombine these two entities by a process of refinement, to retrieve an equivalent concrete CLP program. Finally, the abstract form of the program is used in an abstract analysis of its functional behaviour.

Example 6.1 *Consider the simple CLP program $\Theta = \theta_1, \theta_2, \theta_3$ which is*

$$\forall s, x. (s \geq 5 \wedge x = a) \supset A_1(s, x),$$
$$\forall s, x. (s \geq 9 \wedge \exists y. x = f(y)) \supset A_2(s, x),$$
$$\forall t, x. (\exists y_1, y_2. (\exists s. A_1(s, y_1) \wedge A_2(s, y_2) \wedge t \geq s + 35) \wedge x = g(y_1, y_2)) \supset B(t, x).$$

We imagine Θ as the composition of three components where θ_1, θ_2 each has one output, A_1, A_2 respectively, and no input, while θ_3 has two inputs A_1, A_2 and one output B. The object level parameters s, t represent time and the inequations $s \geq 5$, $s \geq 9$, $t \geq s + 35$ are timing constraints specifying the propagation delays through the components. We read $A_1(s, x)$ as the statement "value x is available at time s on wire A_1." Hence, θ is a component that produces on A_1 the constant value a with an initialization delay of 5 *time units. Similarly, θ_2 is a nondeterministic component that outputs on A_2 all values of the form $f(y)$ for arbitrary y. Finally, θ_3 is an input-output device that computes the value $g(y_1, y_2)$ from its two inputs on A_1 and A_2 with a propagation delay of* 35 *time units.*

Example 6.1 is a very simple program Θ in which two conceptually different aspects of system behaviour are intertwined, in this case function and timing. The timing is captured by inequations and time parameters, while the functionality is represented by abstract predicates A_1, A_2, B. In CLP the difference between these two syntactic elements of Θ can be exploited by declaring the inequation symbol $\geq$ to be a constraint and A_1, A_2, B to be user-defined predicates. In this way the handling of timing inequations is delegated to some built-in constraint solving package, while the generation of these constraints is controlled by executing the abstract user-defined predicates.

6.1 Abstraction

In this section we explain how to form an abstraction $\Theta^{\sharp_2}$ of a concrete CLP program Θ containing constraint formulas B and constraint parameters. The basic idea is simple, almost trivial: Drop all constraint-related parameters, replace every occurrence of an atomic constraint formula B by *true*, and insert $\bigcirc$ at the head of every clause of Θ to indicate that it represents an implication *under some constraint.*

Definition 6.2 *Let $\theta = \forall x_1, \ldots, x_m. S \supset P(x_1, \ldots, x_n)$ be a program clause. A quantifier in θ is called* unobservable *if the variable that is quantified over only occurs in constraint parameters or constraint formulas. Otherwise the quantifier is* observable. *We define $\theta^{\sharp_2}$ as the result of (i) dropping all constraint parameters; (ii) dropping all unobservable quantifiers $\forall x, \exists x$; (iii) replacing all maximal constraint sub-formulas by true, and (iv) replacing the head $P(x_1, \ldots, x_n)$ by $\bigcirc P(x_{\gamma(P)+1}, \ldots, x_n)$. Finally, if S is a Σ-formula then $S^{\sharp_2}$ is obtained from S by transformations (i)–(iii).*

Example 6.3 *Consider the clause θ_3 of example 6.1. We let $t \geq s + 35 \in \Phi_C$ be an atomic constraint predicate and put $\gamma(A_1) = \gamma(A_2) = \gamma(B) = 1$, $\gamma(=) = 0$. We get*

$$\theta_3^{\sharp_2} \quad = \quad \forall x. (\exists y_1, y_2. A_1(y_1) \wedge A_2(y_2) \wedge \mathit{true} \wedge x = g(y_1, y_2)) \ \supset \ \bigcirc B(x).$$

Thus we have eliminated all traces of time and timing constraints. Note that the parameters s, t in $A_1(s, y_1)$, $A_2(s, y_2)$, and $B(t, z)$ disappears from the formula since they are constraint parameter

of the respective relation symbol in each occurrence. Since there are no other occurrences of s or t the quantifiers $\forall t, \exists s$ are unobservable, whence they do not survive the abstraction either.
Using different $\Phi_{\mathcal{C}}$ and $\gamma(\cdot)$ we can obtain various other abstractions of θ_3. For instance, if we take $\gamma(A_1) = \gamma(A_2) = \gamma(=) = 0$ and $\gamma(B) = 1$ with the same $\Phi_{\mathcal{C}}$ we would abstract the timing parameters merely of B, but not of A_1, A_2:

$$\theta_3^{\sharp_2} \quad = \quad \forall x.\, (\exists y_1, y_2.\, (\exists s.\, A_1(s, y_1) \wedge A_2(s, y_2) \wedge \mathit{true}) \wedge x = g(y_1, y_2)) \;\supset\; \bigcirc B(x).$$

This time, the quantor $\exists s$ is observable: s occurring in $A_1(s, y_1)$ is no longer a constraint parameter of A_1 and so the dependency of A_1 on s remains. As another, somewhat extreme, example let us consider both the timing inequation $t \geq s{+}35$ and (in the Herbrand model) the unification constraint $x = g(y_1, y_2)$ as a constraint in $\Phi_{\mathcal{C}}$, and stipulate $\gamma(A_1) = \gamma(A_2) = \gamma(B) = 2$. Then, all parameters are constraint parameters, all quantifiers unobservable, and

$$\theta_3^{\sharp_2} \quad = \quad (A_1 \wedge A_2 \wedge \mathit{true} \wedge \mathit{true}) \;\supset\; \bigcirc B.$$

In this way we abstract from all first-order terms and obtain a purely propositional program. This corresponds to the so-called propositional abstraction *[GW92, Pla81]. The reader may invent other variants of $\theta_3^{\sharp_2}$ by modifying $\Phi_{\mathcal{C}}$ and $\gamma(\cdot)$, for instance with $\gamma(=) = 1$ or $\gamma(=) = 2$.*

Abstracting θ to $\theta^{\sharp_2}$ leaves behind, in general, a number of “dangling” *true* subformulas. With some care the process of abstraction can be refined so as to eliminate[9] these redundancies as well. However, as this would clutter up the presentation of the basic ideas we will not bother to do this here.

Example 6.4 *Let us take the* `mortgage` *program from example 2.1. The program can be presented, modulo a little and innocuous confusion of fonts, as a single clause*

$$\begin{aligned}\mathtt{M} \;:=\; & \forall \mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, \mathtt{D}.\\ & \quad ((\mathtt{D} \leq 1 \wedge \mathtt{B} + \mathtt{MP} = \mathtt{P} * (\mathtt{I} + 1)) \vee \\ & \quad\quad (\mathtt{D} > 1 \wedge \mathtt{mortgage}(\mathtt{P} * (\mathtt{I} + 1) - \mathtt{MP}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, \mathtt{D} - 1))) \;\supset\; \mathtt{mortgage}(\mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, \mathtt{D}).\end{aligned}$$

It will be natural to consider the equation $\mathtt{B} + \mathtt{MP} = \mathtt{P} * (\mathtt{I} + 1)$ as a constraint, and the inequations $\mathtt{D} \leq 1$ and $\mathtt{D} > 1$ as (atomic) program formulas. We put $\gamma(\leq) = \gamma(>) = 0$ and $\gamma(\mathtt{mortgage}) = 4$, so that all four leading parameters $\mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}$ become hidden parameters. Under these circumstances the abstraction of $\mathtt{M}$ is

$$\mathtt{M}^{\sharp_2} \quad = \quad \forall \mathtt{D}.\, ((\mathtt{D} \leq 1 \wedge \mathit{true}) \;\vee\; (\mathtt{D} > 1 \wedge \mathtt{mortgage}(\mathtt{D} - 1))) \;\supset\; \bigcirc\, \mathtt{mortgage}(\mathtt{D}).$$

The constraint sub-formula $\mathtt{B} + \mathtt{MP} = \mathtt{P} * (\mathtt{I} + 1)$ gets abstracted out, as well as the first four parameters of `mortgage`*. Note in particular that the quantifiers $\forall \mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}$ are unobservable. The reader may see what we are getting at: the abstract view $\mathtt{M}^{\sharp_2}$ concentrates on the central aspect of computing* `mortgage`*, which is the recursive iteration through the duration* `D`*. All handling of the arithmetic parameters of principal* `P`*, interest rate* `I`*, monthly payments* `MP`*, balance* `B` *and maintaining of their consistent relationship, thus, is delegated to the constraint system. At least this is the idea. Just how this works in the LLP framework still needs to be seen, of course.*

Now, if the abstraction $\Theta^{\sharp_2}$ is all we do to a program Θ then, clearly, we must lose important information about the semantics of the original program. So we follow our principle of linking constraints with abstractions and record the associated constraints as a sequence of λ-terms $\Theta^{\sharp_1}$ called a *constraint table* for $\Theta^{\sharp_2}$. Later we show how this extra information can be used to recover the constraints from the constraint semantics of $\lambda_c(\mathcal{C})$-proofs obtained from executing the abstract program $\Theta^{\sharp_2}$.

[9] This requires that constraints be closed under disjunctions.

Definition 6.5 *Let $\theta = \forall \vec{x}.\, S \supset P(\vec{y})$ with $\vec{y} \subseteq \vec{x}$ be a program clause. The variables $\vec{x}$ may be partitioned into three disjoint lists: $\vec{x}_o$ denotes the list of variables with observable $\forall$ quantifiers; the remaining $\vec{x}$ variables with unobservable $\forall$ quantifiers we split into two parts, the list $\vec{y}_{no}$ of $\vec{y}$-variables, and the list $\vec{x}_{no}$ of variables different from any $\vec{y}$. Further, let $\vec{y} = y_1, \ldots, y_n$ and $k := \gamma(P) \leq n$. Then, following Def. 6.2, the abstraction of θ is of the shape*

$$\theta^{\sharp_2} \quad = \quad \forall \vec{x}_o.\, S^{\sharp_2} \supset \bigcirc P(y_{k+1}, \ldots, y_n),$$

which by Definition 4.4 has the type $|\theta^{\sharp_2}| = \mathbf{U} \Rightarrow \cdots \Rightarrow \mathbf{U} \Rightarrow |S^{\sharp_2}| \Rightarrow \mathbf{U}^k \Rightarrow \mathbf{C}$. Now we define the constraint table *for θ to be the $\lambda_c(\mathcal{C})$-term*[10]

$$\theta^{\sharp_1} \quad := \quad \lambda \vec{x}_o : \vec{\mathbf{U}}.\, \lambda z : |S^{\sharp_2}|.\, \lambda v : \mathbf{U}^k.\, \exists \vec{x}_{no}.\, \langle S\{\pi_i v / y_i \mid i = 1, \ldots, k \ \&\ y_i \in \vec{y}_{no}\}\rangle_z, \tag{5}$$

of type $|\theta^{\sharp_2}|$, where the constraint term $\langle T\rangle_z : \mathbf{C}$ for every Σ-formula T and $z : |T^{\sharp_2}|$ is obtained by induction on T as follows:

$$\begin{array}{lcl}
\langle B\rangle_z & := & B \\
\langle R(t_1, \ldots, t_r)\rangle_z & := & \pi_1 z = t_1 \wedge \cdots \wedge \pi_k z = t_k \qquad k = \gamma(R) > 0 \\
\langle R(t_1, \ldots, t_r)\rangle_z & := & \mathit{true} \qquad \gamma(R) = 0 \\
\langle T_1 \wedge T_2\rangle_z & := & \langle T_1\rangle_{\pi_1 z} \wedge \langle T_2\rangle_{\pi_2 z} \\
\langle T_1 \vee T_2\rangle_z & := & \mathsf{case}\ z\ \mathsf{of}\ [\iota_1(z_1) \to \langle T_1\rangle_{z_1},\ \iota_2(z_2) \to \langle T_1\rangle_{z_2}] \\
\langle \exists y.\, T\rangle_z & := & \left\{ \begin{array}{ll} \exists y.\, \langle T\rangle_z & \text{if } \exists y \text{ unobservable} \\ \langle T\{\pi_1 z / y\}\rangle_{\pi_2 z} & \text{otherwise} \end{array} \right.
\end{array}$$

Here B is a constraint and $R(t_1, \ldots, t_r)$ is an atomic program formula. It is understood that the last three constructions for $\langle T\rangle$ above only apply if T is not a constraint. In (5) we may have that the set $\{y_i \mid i = 1, \ldots, k \ \&\ y_i \in \vec{y}_{no}\}$ is empty or a single variable.
We put $\theta^\sharp = \theta^{\sharp_1} : \theta^{\sharp_2}$, and if $\Theta = \theta_1, \ldots, \theta_n$ then $\Theta^\sharp$ is $\theta_1^\sharp, \ldots, \theta_n^\sharp$.

The mapping $\theta \mapsto (\theta^{\sharp_1}, \theta^{\sharp_2})$ amounts to a *separation* of two concerns, *viz.* the abstract function $\theta^{\sharp_2}$ and the concrete constraints $\theta^{\sharp_1}$. Taken together both aspects make up the original clause θ, but keeping them separate makes them available for separate treatment so as to formalise the conceptional difference between constraints and formulas. In fact, in our logic framework, constraints $\theta^{\sharp_1}$ behave like proofs whereas the abstract clause $\theta^{\sharp_2}$ behaves like a formula. This suggests the *proof : formula* notation $\theta^{\sharp_1} : \theta^{\sharp_2}$. In the following every $\lambda_c(\mathcal{C})$-term p of type θ is called a *constraint table* for θ.

Example 6.6 *Consider the clause θ_1 of Example 6.1 with $s \geq 5 \in \Phi_{\mathcal{C}}$, $\gamma(A_1) = 1$ and $\gamma(=) = 0$. We compute the constraint table for θ_1 according to Def. 6.5 as follows:*

$$\begin{array}{rcl}
\theta_1^{\sharp_1} & = & (\forall s, x.\, (s \geq 5 \wedge x = a) \supset A_1(s, x))^{\sharp_1} \\
& = & \lambda x.\, \lambda z.\, \lambda v.\, \langle (v \geq 5 \wedge x = a)\rangle_z \\
& = & \lambda x.\, \lambda z.\, \lambda v.\, \langle v \geq 5\rangle_{\pi_1 z} \wedge \langle x = a\rangle_{\pi_2 z} \\
& = & \lambda x.\, \lambda z.\, \lambda v.\, v \geq 5 \wedge \mathit{true}.
\end{array}$$

It is not difficult to verify that this $\lambda_c(\mathcal{C})$-term indeed has type $|\theta_1^{\sharp_2}| = |\forall x.\, (\mathit{true} \wedge x = a) \supset \bigcirc A_1(x)| = \mathbf{U} \Rightarrow (\mathbf{1} \times \mathbf{1}) \Rightarrow \mathbf{U} \Rightarrow \mathbf{C}$. Now, if we write down both $\theta_1^{\sharp_1}$ and $\theta_1^{\sharp_2}$ side-by-side

$$\theta_1^\sharp \ = \ \theta_1^{\sharp_1} : \theta_1^{\sharp_2} \ = \ \lambda x.\, \lambda z.\, \lambda v.\, v \geq 5 \wedge \mathit{true} : \forall x.\, (\mathit{true} \wedge x = a) \supset \bigcirc A_1(x),$$

[10] The notation $\lambda \vec{x}_o : \vec{\mathbf{U}}$, which stands for a sequence of λ-abstractions, is slightly inaccurate but clear enough.

and compare this with θ_1 we can see how by this process the original concrete clause θ_1 has been reorganised into two parts, a constraint table to the left of the colon and a remaining abstracted clause to the right. Both sides considered together contain exactly the same information as the original concrete clause. How θ_1 can be reconstructed from this pair, up to logic equivalence, by a systematic refinement will be discussed in the next section.

Example 6.7 *For the following discussions it will be convenient to simplify our running example 6.1 in that we drop the "functional" aspects and consider only the "timing" behaviour. More precisely, we consider the abridged program $\Theta = \theta_1, \theta_2, \theta_3$ which is*

$$\begin{gathered}\forall s.\, s \geq 5 \supset A_1(s),\\ \forall s.\, s \geq 9 \supset A_2(s),\\ \forall t.\, (\exists s.\, A_1(s) \wedge A_2(s) \wedge t \geq s + 35) \supset B(t).\end{gathered}$$

This simplification itself can, in fact, be considered as a formal abstraction in the sense of our Def. 6.2 except that it is performed at the meta-level "manually" and thus we do not need to bother with the modality. In effect, we forget the functionality completely rather than recording it in a constraint table.[11] *We now wish to form $\Theta^\sharp$ from the concrete program Θ with $e_1 \geq e_2 \in \Phi_\mathcal{C}$ and $\gamma(A_1) = \gamma(A_2) = \gamma(B) = 1$. For this, both $\Theta^{\sharp_1} = \theta_1^{\sharp_1}, \theta_2^{\sharp_1}, \theta_3^{\sharp_1}$ and $\Theta^{\sharp_2} = \theta_1^{\sharp_2}, \theta_2^{\sharp_2}, \theta_3^{\sharp_2}$ are required. Let us consider $\Theta^{\sharp_2}$ first. Definition 6.2 gives us*

$$\begin{aligned}\theta_1^{\sharp_2} &:= \quad \mathit{true} \supset \bigcirc A_1\\ \theta_2^{\sharp_2} &:= \quad \mathit{true} \supset \bigcirc A_2\\ \theta_3^{\sharp_2} &:= \quad (A_1 \wedge A_2 \wedge \mathit{true}) \supset \bigcirc B\end{aligned}$$

Also, for each clause of the concrete program Θ, we must determine the corresponding $\lambda_c(\mathcal{C})$-term. Using Definition 6.5 gives us

$$\begin{aligned}\theta_1^{\sharp_1} &= \quad (\forall s.\, s \geq 5 \supset A_1(s))^{\sharp_1} = \lambda z.\, \lambda v.\, v \geq 5\\ \theta_2^{\sharp_1} &= \quad (\forall s.\, s \geq 9 \supset A_2(s))^{\sharp_1} = \lambda z.\, \lambda v.\, v \geq 9\\ \theta_3^{\sharp_1} &= \quad (\forall t.\, (\exists s.\, A_1(s) \wedge A_2(s) \wedge t \geq s + 35) \supset B(t))^{\sharp_1}\\ &\quad\; = \lambda z.\, \lambda v.\, \exists s.\, \pi_1 z = s \wedge \pi_2 z = s \wedge v \geq s + 35.\end{aligned}$$

Thus we have defined $\theta_i^{\sharp_1} : \theta_i^{\sharp_2}$ for each $i \in \{1, 2, 3\}$, and therefore $\Theta^\sharp$. Notice, the constraint table for θ_3 with the extracted constraint $\exists s.\, \pi_1 z = s \wedge \pi_2 z = s \wedge v \geq s + 35$: There is not just mention of the timing inequation. Also there is the conjunction $\pi_1 z = s \wedge \pi_2 z = s$ which, on the face of it, does not seem to correspond to any syntactic part of θ_3. Yet, it corresponds to an important constraint, viz. the implicit constraint that arises from the fact that the two conjuncts $A_1(s) \wedge A_2(s)$ in the body of clause θ_3 share the same variable s. These "sharing" constraints are automatically taken care of by our abstraction method.

Example 6.8 *What is the constraint table for the abstracted clause*

$$\mathtt{M}^{\sharp_2} \quad = \quad \forall \mathtt{D}.\, ((\mathtt{D} \leq \mathtt{1} \wedge \mathit{true}) \;\vee\; (\mathtt{D} > \mathtt{1} \wedge \mathtt{mortgage}(\mathtt{D} - \mathtt{1}))) \;\supset\; \bigcirc\mathtt{mortgage}(\mathtt{D})$$

of program `mortgage`*? The constraint table will have the type $|\mathtt{M}^{\sharp_2}|$ which is $\mathbf{U} \Rightarrow ((\mathbf{1} \times \mathbf{1}) + (\mathbf{1} \times \mathbf{U}^4)) \Rightarrow \mathbf{U}^4 \Rightarrow \mathbf{C}$. Following Definition 6.5 the constraint table $\mathtt{M}^{\sharp_1}$ is*

[11] We conjecture that by a natural generalisation of our method abstractions can be cascaded.

$\lambda\mathtt{D},\mathtt{z},\mathtt{v}.\,\langle \mathtt{Mbody}\{\pi_1\mathtt{v},\pi_2\mathtt{v},\pi_3\mathtt{v},\pi_4\mathtt{v}/\mathtt{P},\mathtt{I},\mathtt{MP},\mathtt{B}\}\rangle_{\mathtt{z}}$ *where* Mbody *is the body of clause* M. *We find:*

$$
\begin{aligned}
&\langle \mathtt{Mbody}\{\pi_1\mathtt{v},\pi_2\mathtt{v},\pi_3\mathtt{v},\pi_4\mathtt{v}/\mathtt{P},\mathtt{I},\mathtt{MP},\mathtt{B}\}\rangle_{\mathtt{z}}\\
&\quad= \langle\, ((\mathtt{D} \le 1 \land \pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1)) \lor \\
&\qquad\quad (\mathtt{D} > 1 \land \mathtt{mortgage}(\pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v}, \pi_2\mathtt{v}, \pi_3\mathtt{v}, \pi_4\mathtt{v}, \mathtt{D}-1)))\,\rangle_{\mathtt{z}}\\
&\quad= \mathsf{case}\ \mathtt{z}\ \mathsf{of}\ [\iota_1(\mathtt{z}_1) \to \langle\, \mathtt{D} \le 1 \land \pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1)\,\rangle_{\mathtt{z}_1},\\
&\qquad\qquad\qquad\quad \iota_2(\mathtt{z}_2) \to \langle\, \mathtt{D} > 1 \land \mathtt{mortgage}(\pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v}, \pi_2\mathtt{v}, \pi_3\mathtt{v}, \pi_4\mathtt{v}, \mathtt{D}-1)\,\rangle_{\mathtt{z}_2}]\\
&\quad= \mathsf{case}\ \mathtt{z}\ \mathsf{of}\ [\iota_1(\mathtt{z}_1) \to \langle \mathtt{D} \le 1\rangle_{\pi_1\mathtt{z}_1} \land \langle \pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1)\rangle_{\pi_2\mathtt{z}_1},\\
&\qquad\qquad\qquad\quad \iota_2(\mathtt{z}_2) \to \langle \mathtt{D} > 1\rangle_{\pi_1\mathtt{z}_2} \land \langle \mathtt{mortgage}(\pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v}, \pi_2\mathtt{v}, \pi_3\mathtt{v}, \pi_4\mathtt{v}, \mathtt{D}-1)\rangle_{\pi_2\mathtt{z}_2}]\\
&\quad= \mathsf{case}\ \mathtt{z}\ \mathsf{of}\ [\iota_1(\mathtt{z}_1) \to \mathit{true} \land \pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1),\\
&\qquad\qquad\qquad\quad \iota_2(\mathtt{z}_2) \to \mathit{true} \land \pi_1\pi_2\mathtt{z}_2 = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v} \land\\
&\qquad\qquad\qquad\qquad\qquad \pi_2\pi_2\mathtt{z}_2 = \pi_2\mathtt{v} \land \pi_3\pi_2\mathtt{z}_2 = \pi_3\mathtt{v} \land \pi_4\pi_2\mathtt{z}_2 = \pi_4\mathtt{v}].
\end{aligned}
$$

Hence, in total,

$$
\begin{aligned}
\mathtt{M}^{\sharp_1} \;=\; \lambda\mathtt{D},\mathtt{z},\mathtt{v}.\,\mathsf{case}\ \mathtt{z}\ \mathsf{of}\ [&\iota_1(\mathtt{z}_1) \to \mathit{true} \land \pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1),\\
&\iota_2(\mathtt{z}_2) \to \mathit{true} \land \pi_1\pi_2\mathtt{z}_2 = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v} \land\\
&\qquad \pi_2\pi_2\mathtt{z}_2 = \pi_2\mathtt{v} \land \pi_3\pi_2\mathtt{z}_2 = \pi_3\mathtt{v} \land \pi_4\pi_2\mathtt{z}_2 = \pi_4\mathtt{v}].
\end{aligned}
$$

Here the equation $\pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1)$ *internalises the constraint* $\mathtt{B} + \mathtt{MP} = \mathtt{P} * (\mathtt{I}+1)$ *of* mortgage, *the equation* $\pi_1\pi_2\mathtt{z}_2 = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v}$ *captures the pattern* $\mathtt{P} * (\mathtt{I}+1) - \mathtt{MP}$, *and* $\pi_2\pi_2\mathtt{z}_2 = \pi_2\mathtt{v} \land \pi_3\pi_2\mathtt{z}_2 = \pi_3\mathtt{v} \land \pi_4\pi_2\mathtt{z}_2 = \pi_4\mathtt{v}$ *the other hidden parameters in the recursive call* $\mathtt{mortgage}(\mathtt{P} * (\mathtt{I}+1) - \mathtt{MP}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, \mathtt{D}-1)$ *of the original program from Example 2.1. Since this example is about to turn a bit bulky notationally let us agree to use some abbreviations. We write* m *rather than* morgage *and short-name the relevant constraints as follows:*

$$
\begin{aligned}
\mathtt{c}_1(\mathtt{v}) &:= \mathit{true} \land \pi_4\mathtt{v} + \pi_3\mathtt{v} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1)\\
\mathtt{c}_2(\mathtt{v},\mathtt{z}) &:= \pi_1\pi_2\mathtt{z} = \pi_1\mathtt{v} * (\pi_2\mathtt{v}+1) - \pi_3\mathtt{v} \land \pi_2\pi_2\mathtt{z} = \pi_2\mathtt{v} \land \pi_3\pi_2\mathtt{z} = \pi_3\mathtt{v} \land \pi_4\pi_2\mathtt{z} = \pi_4\mathtt{v}.
\end{aligned}
$$

Using these abbreviations the abstracted program and its associated constraint table can be written more succinctly as

$$
\begin{aligned}
\mathtt{M}^{\sharp_2} &= \forall\mathtt{D}.\,((\mathtt{D} \le 1 \land \mathit{true}) \;\lor\; (\mathtt{D} > 1 \land \mathtt{m}(\mathtt{D}-1))) \;\supset\; \bigcirc\mathtt{m}(\mathtt{D})\\
\mathtt{M}^{\sharp_1} &= \lambda\mathtt{D},\mathtt{z},\mathtt{v}.\,\mathsf{case}\ \mathtt{z}\ \mathsf{of}\ [\iota_1(\mathtt{z}_1) \to \mathtt{c}_1(\mathtt{v}), \iota_2(\mathtt{z}_2) \to \mathtt{c}_2(\mathtt{v},\mathtt{z}_2)].
\end{aligned}
$$

The following theorem shows that the process of constraint abstraction may be mirrored at the level of proofs. We can use abstraction to clean up with constraints and turn a partial proof into a complete proof.

Theorem 6.9 (Soundness of Abstraction) *Let* Θ *be a CLP-program and* S *a* Σ*-formula. For every partial proof* $\Theta \vdash_{\mathrm{r}} S$ *there is a complete proof* $\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc S^{\sharp_2}$*, and for every* $\Theta, B_1, \ldots, B_n \vdash_{\mathrm{l}} S$*, with* B_i $(i = 1, \ldots, n)$ *constraints, there is a proof* $\Theta^{\sharp_2} \vdash_{\mathrm{l}} \bigcirc S^{\sharp_2}$.

To be a bit more concrete, suppose $\Theta = \theta_1, \ldots, \theta_n$ is a CLP-program, $P(\vec{x})$ a query, and $B_1, \ldots, B_n$ constraint formulas such that $\Theta, B_1, \ldots, B_n \vdash_{\mathrm{l}} P(\vec{x})$ is derivable. We can interpret $B_1 \land \cdots \land B_n$ as the answer constraint for query P on program Θ. Theorem 6.9 now means that we can drop all constraints and constraint parameters, and derive the constraint-free sequent $\Theta^{\sharp_2} \vdash_{\mathrm{l}} \bigcirc P(t_{k+1}, \ldots, t_m)$ where $k = \gamma(P)$. We will see that the constraint B can be viewed as being generated from the abstract proof by systematic refinement.

Example 6.10 *Let us take a look at the simplified CLP program $\Theta = \theta_1, \theta_2, \theta_3$ as in Example 6.7. We can attempt to prove the query $B(t)$ from Θ, which of course, we cannot without accumulating some proof obligations concerning timing constraints. We use LLP as an operational semantics for Θ, construct a partial proof and leave these timing constraints as open leaves. Such a proof tree may look like:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Theta \vdash_{\mathrm{r}} s \geq 5}{\Theta \vdash_{\mathrm{r}} A_1(s)}\supset_{\mathcal{N}} \quad \dfrac{\Theta \vdash_{\mathrm{r}} s \geq 9}{\Theta \vdash_{\mathrm{r}} A_2(s)}\supset_{\mathcal{N}}}{\Theta \vdash_{\mathrm{r}} A_1(s) \wedge A_2(s)}\wedge_{\mathcal{I}} \qquad \Theta \vdash_{\mathrm{r}} t \geq s + 35}{\Theta \vdash_{\mathrm{r}} A_1(s) \wedge A_2(s) \wedge t \geq s + 35}\wedge_{\mathcal{I}}}{\Theta \vdash_{\mathrm{r}} \exists s.\, A_1(s) \wedge A_2(s) \wedge t \geq s + 35}\exists_{\mathcal{I}}}{\Theta \vdash_{\mathrm{r}} B(t)}\supset_{\mathcal{N}}$$

Note that the proof tree indeed is a partial proof since all leaves are sequents of form $\Theta \vdash_{\mathrm{r}} B$ with B a constraint. These leaves represent a very special kind of dangling proof obligations in the sense that they must not be taken too literally as place holders for yet-to-be-instantiated proofs. For, in fact, the proof tree cannot be completed as this would involve proving unreasonable facts about constraints, even if we had available a complete theory of timing constraints. The open proof leaf $\Theta \vdash_{\mathrm{r}} t \geq s + 35$, for instance, is logically the same as $\Theta \vdash \forall s, t.\, t \geq s + 35$, which is plainly inconsistent with the theory of natural numbers. But then, so we must ask, what is this partial proof good for if it cannot be completed? The answer is that it is a complete proof at the abstract level. The abstraction applies $(\cdot)^{\sharp_2}$ to all formulas, in the hypothesis and conclusion of the sequents, and eliminates unobservable proof rules. For the proof tree of this example we obtain the following abstracted tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc true}true_{\bigcirc}}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc A_1}\supset_{\bigcirc} \quad \dfrac{\dfrac{}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc true}true_{\bigcirc}}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc A_2}\supset_{\bigcirc}}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc(A_1 \wedge A_2)}\wedge_{\bigcirc} \qquad \dfrac{}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc true}true_{\bigcirc}}{\dfrac{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc(A_1 \wedge A_2 \wedge true)}{\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc B}\supset_{\bigcirc}}\wedge_{\bigcirc}$$

This is a proper LLP-proof without any open proof obligations. According to $(\cdot)^{\sharp_2}$ the dangling constraint sequents $\Theta \vdash_{\mathrm{r}} C$ are replaced by $\Theta^{\sharp_2} \vdash_{\mathrm{r}} \bigcirc true$, which can be discharged by rule $true_{\bigcirc}$. Further the proof rules $\wedge_{\mathcal{I}}, \supset_{\mathcal{N}}$ of the previous CLP derivation become the derived rules $\wedge_{\bigcirc}, \supset_{\bigcirc}$. The rule $\exists_{\mathcal{I}}$ disappears since the quantifier $\exists s$ is unobservable. Sure enough, this abstraction process has not solved any constraint but it has not lost any information either. The constraints are still there though not in the propositions but in the proofs.

Example 6.11 *We can also abstract $\vdash_{\mathrm{l}}$ derivations. A concrete CLP-derivation for the previous Example 6.10 runs as follows:*

$$\begin{aligned}
&\rightsquigarrow\ \Theta, B(t) \vdash_{\mathrm{l}} B(t) \ \rightsquigarrow\ \Theta, \exists s.\, A_1(s) \wedge A_2(s) \wedge t \geq s + 35 \vdash_{\mathrm{l}} B(t) \\
&\quad \rightsquigarrow\ \Theta, A_1(s) \wedge A_2(s) \wedge t \geq s + 35 \vdash_{\mathrm{l}} B(t) \ \rightsquigarrow\ \Theta, A_1(s) \wedge A_2(s), t \geq s + 35 \vdash_{\mathrm{l}} B(t) \\
&\qquad \rightsquigarrow\ \Theta, A_1(s), A_2(s), t \geq s + 35 \vdash_{\mathrm{l}} B(t) \ \rightsquigarrow\ \Theta, s \geq 5, A_2(s), t \geq s + 35 \vdash_{\mathrm{l}} B(t) \\
&\qquad\quad \rightsquigarrow\ \Theta, s \geq 5, s \geq 9, t \geq s + 35 \vdash_{\mathrm{l}} B(t)
\end{aligned}$$

where each $\leadsto$ *indicates a single application of a* $\vdash_{\mathrm{l}}$ *rule. The abstraction, which ignores constraints and constraint parameters, is*

$$\begin{aligned}&\leadsto\ \Theta, B \vdash_{\mathrm{l}} \bigcirc B \ \leadsto\ \Theta, A_1 \wedge A_2 \wedge \mathit{true} \vdash_{\mathrm{l}} \bigcirc B \ \leadsto\ \Theta, A_1 \wedge A_2, \mathit{true} \vdash_{\mathrm{l}} \bigcirc B\\ &\leadsto\ \Theta, A_1, A_2, \mathit{true} \vdash_{\mathrm{l}} \bigcirc B \ \leadsto\ \Theta, \mathit{true}, A_2, \mathit{true} \vdash_{\mathrm{l}} \bigcirc B \ \leadsto\ \Theta, \mathit{true}, \mathit{true}, \mathit{true} \vdash_{\mathrm{l}} \bigcirc B\\ &\leadsto\ \Theta, \mathit{true}, \mathit{true} \vdash_{\mathrm{l}} \bigcirc B \ \leadsto\ \Theta, \mathit{true} \vdash_{\mathrm{l}} \bigcirc B \ \leadsto\ \Theta \vdash_{\mathrm{l}} \bigcirc B.\end{aligned}$$

We now show what the abstraction looks like at the level of proofs, for $\vdash_{\mathrm{r}}$ derivations.

Definition 6.12 *Let* Θ *be a CLP program,* S *a* Σ*-formula, and* $\vec{w} : \Theta \vdash_{\mathrm{r}} p : S$ *be a partial proof. We obtain the abstracted proof* $\vec{w} : \Theta^{\sharp_2} \vdash_{\mathrm{r}} p^{\sharp_1} : \bigcirc S^{\sharp_2}$ *by induction on* p *as follows:*

$$\begin{array}{rcll} *^{\sharp_1} & := & \mathit{true}_{\bigcirc} & \\ ?^{\sharp_1} & := & \mathit{true}_{\bigcirc} & \\ (p,q)^{\sharp_1} & := & \wedge_{\bigcirc}(p^{\sharp_1}, q^{\sharp_1}) & \\ (\iota_i p)^{\sharp_1} & := & \vee_{\bigcirc}(p^{\sharp_1}, i) & i = 1,2 \\ (\supset_{\mathcal{N}}(p, w, t_1, \ldots, t_n))^{\sharp_1} & := & \supset_{\bigcirc}(p^{\sharp_1}, w, t_{o_1}, \ldots, t_{o_l}) & \\ \multicolumn{4}{l}{\quad \textit{where } w : \forall\vec{x}.\, S \supset P(\vec{y}) \in \Theta, \textit{ and } \forall x_{o_1}, \ldots, x_{o_l} \textit{ is the list of observable quantifiers of } \forall\vec{x}} \\ (\iota_t p)^{\sharp_1} & := & \exists_{\bigcirc}(p^{\sharp_1}, t) & \textit{if } \exists x \textit{ is observable} \\ (\iota_t p)^{\sharp_1} & := & p^{\sharp_1} & \textit{if } \exists x \textit{ unobservable,} \end{array}$$

where $?$ *is a dangling leaf* $\Pi \vdash_{\mathrm{r}} ? : B$ *of the partial proof* $\vec{w} : \Pi \vdash_{\mathrm{r}} p : S$.

Example 6.13 *Suppose that in the proof of Example 6.10 the program clauses are referred to by the proof variables* $\vec{w} : \Theta = w_1 : \theta_1, w_2 : \theta_2, w_3 : \theta_3$. *Then, the CLP proof term associated with the first proof tree of 6.10 is* $p = w_3 t(\iota_s(w_1 s?, w_2 s?, ?)) : B(t)$. *The abstraction is*

$$p^{\sharp_1} = \supset_{\bigcirc}(\wedge_{\bigcirc}(\wedge_{\bigcirc}(\supset_{\bigcirc}(\mathit{true}_{\bigcirc}, w_1), \supset_{\bigcirc}(\mathit{true}_{\bigcirc}, w_2)), \mathit{true}_{\bigcirc}), w_3)$$

which is nothing but the proof term obtained from the second, abstracted, proof tree of Example 6.10.

6.2 Refinement

The question we wish to address in this section is what is the meaning of a pair $\Xi : \Pi$ where Ξ is a constraint table and Π an abstract program. This meaning will be given by a syntactic translation that turns the *proof : formula* pair into a proper CLP-program. This translation will justify our claim that $\Xi : \Pi$ is a 'refinement' of program Π by the extra intensional information Ξ. In essence, this refinement $(\Xi : \Pi)^\flat$ will be set up such that it reverses the constraint abstraction and recombines an abstract program Π with a concrete constraint table Ξ.

For every Σ-formula S the elements of $|S|$ are *formal indices* for accessing the constraints in a constraint table for an abstract program clause $\zeta = \forall\vec{x}.\ S \supset \bigcirc P(\vec{y})$, that has S as its body. To be more precise suppose p is a constraint table for ζ. Then, for every formal index $g : |S|$, the corresponding entry of the constraint table p at g is represented by the term $p\,\vec{x}\,g : \mathbf{U}^k \Rightarrow \mathbf{C}$, where $k = \gamma(P)$, which is a constraint on the hidden variables of P. In our prospective refinement $(p : \zeta)^\flat$ we wish to reintroduce this constraint into the right place of the clause's body S. The right place is obtained by computing a refinement $(g : S)^\flat$ which picks out the part of S that corresponds to index g. Assuming this is done, we obtain $(p : \zeta)^\flat$ then formally as

$$\forall\vec{z}.\forall\vec{x}. \quad (\exists g : |S|.\, (g : S)^\flat \wedge p\,\vec{x}\,g\,(z_1, \ldots, z_k)) \ \supset\ P(\vec{z}, \vec{y}) \tag{6}$$

where $\vec{z} = z_1, \ldots, z_k$ is a list of fresh variables for the hidden parameters of P. The body of this refined clause, roughly speaking, can be read as follows: First, the abstract body S is projected to its indices g, by $(g : S)^\flat$; Then the constraint relevant for this index is imposed, which gives $(g : S)^\flat \wedge p\,\vec{x}\,g\,(z_1, \ldots, z_k)$; Finally, the total body is reconstructed by existentially quantifying over all indices. This yields the new body $\exists g : |S|.\,(g : S)^\flat \wedge p\,\vec{x}\,g\,(z_1, \ldots, z_k)$ in the refined clause (6). Note that the constraint $p\,\vec{x}\,g\,(z_1, \ldots, z_k)$ at every index g may depend both on the free variables $\vec{x}$ of the abstract clause (the "observable" variables) and the hidden variables $\vec{z}$ (the "unobservable" variables) of P. Of course, not only the hidden variables of P must be made explicit by the refinement but also all other hidden variables pertaining to the atomic formulas in S. How this is done will be seen shortly.

In fact, we must be a little bit more careful. For even if we have a suitable definition of $(g : S)^\flat$, the expression (6), though it looks very much like a (refined) program clause, is not (yet) a proper formula of our logic. Problem number one is that the subexpression $p\,\vec{x}\,g\,(z_1, \ldots, z_k)$ really is a $\lambda_c(\mathcal{C})$-term of type $\mathbf{C}$ as opposed to a proper constraint formula. However, assuming that g is a pseudo-closed $\lambda_c(\mathcal{C})$-term, this problem can be solved simply by reduction (see Prop. 4.7). Problem number two is the fact that we do not have in our first-order logic an object level type $|S|$ so that the quantification $\exists g : |S|$ in (6) does not make sense as it stands. Since $|S|$ has an infinite number of elements, *e.g.* quasi-closed normal form terms, we cannot in general trivially eliminate $\exists g : |S|$ completely by a finite disjunction. Fortunately, it turns out that the quantifier can be eliminated in terms of a finite number of existential quantifications over first-order variables and a finite number of explicit disjunctions. This is good enough.

Hence a bit of extra work must be done to turn (6) into a proper QLL formula which thus can be fed back into our logic. All in all, a refinement $(p : \theta)^\flat$ of an abstract program clause ζ by a constraint table p will involve three processes: *quantifier elimination*, *index projection*, and *reduction*. Since we already know how to reduce $\lambda_c(\mathcal{C})$-terms we only need to define the quantifier elimination and index projection in the following.

Definition 6.14 (Index Projection) *Let S be a Σ-formula and $g : |S|$ a quasi-closed normal form $\lambda_c(\mathcal{C})$-term. We define the refinement $(g : S)^\flat$ by induction on S as follows:*

$$
\begin{aligned}
((s_1, \ldots, s_k) : R(t_1, \ldots, t_n))^\flat &:= R(s_1, \ldots, s_k, t_1, \ldots, t_n) \\
(* : R(t_1, \ldots, t_n))^\flat &:= R(t_1, \ldots, t_n) \\
(* : \mathit{true})^\flat &:= \mathit{true} \\
((g_1, g_2) : S_1 \wedge S_2)^\flat &:= (g_1 : S_1)^\flat \wedge (g_2 : S_2)^\flat \\
(\iota_i g : S_1 \vee S_2)^\flat &:= (g : S_i)^\flat \qquad i \in \{1, 2\} \\
((t, g) : \exists y.\, S)^\flat &:= (g : S\{t/y\})^\flat,
\end{aligned}
$$

where in the first line $k = \gamma(R) > 0$ and in the second $k = \gamma(R) = 0$.

Example 6.15 *Let $S = (A_1 \vee A_2) \wedge A_3$ and assume $\gamma(A_i) = 0$. Then, $|S| = (\mathbf{1} + \mathbf{1}) \times \mathbf{1}$. There are exactly two quasi-closed normal form terms of this type, viz. $g_1 = (\iota_1 *, *)$ and $g_2 = (\iota_2 *, *)$, which gives two formal indices for S. Applying index projection obtains $(g_1 : S)^\flat = ((\iota_1 *, *) : (A_1 \vee A_2) \wedge A_3)^\flat = (\iota_1 * : A_1 \vee A_2)^\flat \wedge (* : A_3)^\flat = (* : A_1)^\flat \wedge (* : A_3)^\flat = A_1 \wedge A_3$. Similarly, the other projection is $(g_2 : S)^\flat = A_2 \wedge A_3$. We notice that up to logic equivalence S can be reconstructed from its two projections, viz. $S \equiv (g_1 : S)^\flat \vee (g_2 : S)^\flat \equiv (A_1 \wedge A_3) \vee (A_2 \wedge A_3)$. This is a general fact: Suppose S does not contain quantifiers and $\gamma(R) = 0$ for all relation symbols R occurring in S. Then there exist only a finite number of quasi-closed normal form terms of type $|S| \cong \mathbf{1} + \cdots + \mathbf{1} \cong n \cdot \mathbf{1}$. If $\mathsf{ind}(S)$ denotes this finite set of formal indices, then S can be decomposed as $S \equiv \bigvee_{g \in ind(S)} (g : S)^\flat$. This decomposition is nothing but the disjunctive normal form of S.*

Example 6.16 *When S contains quantifiers the decomposition by indices needs more thought. Let $S = ((\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2)) \wedge A_3$, and again $\gamma(A_i) = 0$. Now $|S| = ((\mathbf{U} \times \mathbf{1}) + (\mathbf{U} \times \mathbf{1})) \times \mathbf{1}$ which has an infinite number of elements. The quasi-closed normal form terms of type $|S|$ are of the form $g_i(t) = (\iota_i(t, *), *)$ where $i \in \{1, 2\}$ and t is an arbitrary object-level term. We obtain $(g_i(t) : S)^\flat = A_i(t) \wedge A_3$. Obviously, this time the disjunctive decomposition $S \equiv \bigvee_{i,t} A_i(t) \wedge A_3$ is unreasonable since, for one, formulas must be finite and, for another, the infinite disjunction would only take care of the original quantifications $\exists y_i.\, A_i(y_i)$ for elements of the universe that can actually be expressed by terms t in the language. The right disjunctive decomposition of the form $S \equiv \exists g : |S|.\, (g : S)^\flat$ is $S \equiv (\exists y_1.\, A_1(y_1) \wedge A_3) \vee (\exists y_2.\, A_2(y_2) \wedge A_3)$. We will construct this systematically by a method of quantifier elimination (see Def. 6.17).*

Definition 6.17 (Quantifier Elimination) *Let S be a Σ-formula and $M[g]$ a family of arbitrary formulas indexed in g, where g ranges over all quasi-closed normal form $\lambda_c(\mathcal{C})$-terms of type $|S|$. We define the formula*

$$\exists z : |S|.\, M[z]$$

as an abbreviation in the following way by induction on the structure of S:

$$\begin{aligned}
\exists z : |\mathit{true}|.\, M[z] &:= M[*] \\
\exists z : |R(t_1, \ldots, t_n)|.\, M[z] &:= \exists x_1, \ldots, x_k.\, M[(x_1, \ldots, x_k)] \qquad \text{if } k = \gamma(R) > 0 \\
\exists z : |R(t_1, \ldots, t_n)|.\, M[z] &:= M[*] \qquad \text{if } k = \gamma(R) = 0 \\
\exists z : |S_1 \wedge S_2|.\, M[z] &:= \exists z_1 : |S_1|.\, \exists z_2 : |S_2|.\, M[(z_1, z_2)] \\
\exists z : |S_1 \vee S_2|.\, M[z] &:= (\exists z_1 : |S_1|.\, M[\iota_1 z_1]) \ \vee \ (\exists z_2 : |S_2|.\, M[\iota_2 z_2]) \\
\exists z : |\exists y.\, S|.\, M[z] &:= \exists y.\, \exists z : |S|.\, M[(y, z)].
\end{aligned}$$

Example 6.18 *Take $S = ((\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2)) \wedge A_3$ as in Example 6.16. We compute $\exists z : |S|.\, (z : S)^\flat$ by quantifier elimination (Definition 6.17):*

$$\begin{aligned}
\exists z : |S|.\, (z : S)^\flat &= \exists z : |((\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2)) \wedge A_3|.\, (z : S)^\flat \\
&= \exists z_1 : |(\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2)|.\, \exists z_2 : |A_3|.\, ((z_1, z_2) : S)^\flat \\
&= \exists z_1 : |(\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2)|.\, ((z_1, *) : S)^\flat \\
&= \textstyle\bigvee_{i=1}^2 \exists z_i : |\exists y_i.\, A_i(y_i)|.\, ((\iota_i z_i, *) : S)^\flat \\
&= \textstyle\bigvee_{i=1}^2 \exists y_i.\, \exists z_i : |A_i(y_i)|.\, ((\iota_i(y_i, z_i), *) : S)^\flat \\
&= \textstyle\bigvee_{i=1}^2 \exists y_i.\, ((\iota_i(y_i, *), *) : S)^\flat \\
&= \cdots
\end{aligned}$$

and from there further with index projection (Definition 6.14):

$$\begin{aligned}
\cdots &= \textstyle\bigvee_{i=1}^2 \exists y_i.\, ((\iota_i(y_i, *), *) : ((\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2)) \wedge A_3)^\flat \\
&= \textstyle\bigvee_{i=1}^2 \exists y_i.\, (\iota_i(y_i, *) : (\exists y_1.\, A_1(y_1)) \vee \exists y_2.\, A_2(y_2))^\flat \ \wedge \ (* : A_3)^\flat \\
&= \textstyle\bigvee_{i=1}^2 \exists y_i.\, ((y_i, *) : \exists y_i.\, A_i(y_i))^\flat \ \wedge \ A_3 \\
&= \textstyle\bigvee_{i=1}^2 \exists y_i.\, (* : A_i(y_i))^\flat \ \wedge \ A_3 \\
&= \textstyle\bigvee_{i=1}^2 \exists y_i.\, A_i(y_i) \ \wedge \ A_3.
\end{aligned}$$

Thus, $\exists z : |S|.\, (z : S)^\flat = (\exists y_1.\, A_1(y_1) \wedge A_3) \vee (\exists y_2.\, A_2(y_2) \wedge A_3)$, which is equivalent to S.
It may be remarked that in order for our derivation to be strictly in line with the definitions we should first apply index projection to $(g : S)^\flat$ for all formal indices g to find out about the family $S^\flat[g] = (g : S)^\flat$, and then apply quantifier elimination to compute $\exists z : |S|.\, S^\flat[z]$ from this family.

Yet, as often in constraint programming, here too it is more convenient to start from the end and do the necessary constructions on demand.

Example 6.19

The examples indicate what we have proposed before, *viz.* that any Σ-formula can be decomposed into a disjunctive normal form using its formal indices.

Proposition 6.20 *Suppose that $\gamma(R) = 0$ for all relation symbols R. Then, for every Σ-formula S we have the equivalence $S \equiv \exists z : |S|.\, S^\flat[z]$, obtained from the family $S^\flat[g] = (g : S)^\flat$ where g ranges over all quasi-closed normal form terms $g : |S|$. We may write this equivalence more compactly as $S \equiv \exists z : |S|.\, (g : S)^\flat$. In general, we have $S' \equiv \exists z : |S|.\, (g : S)^\flat$, where S' is obtained from S by replacing every occurrence $P(\vec{t})$ of an atomic program formula by $\exists x_1, \ldots, x_k.\, P(\vec{x}, \vec{t})$ where $k = \gamma(P)$. All equivalences are provable in QLL.*

We can now finalise our notion of program refinement.

Definition 6.21 (Program Refinement) *Let $\zeta = \forall \vec{x}.\ S \supset \bigcirc P(\vec{y})$, $\vec{y} \subseteq \vec{x}$, be an abstract program clause and p a constraint table for ζ. Let $k := \gamma(P)$. We define the* refinement *of ζ* by *p as the modal-free formula*

$$(p : \zeta)^\flat \quad := \quad \forall \vec{z}. \forall \vec{x}. (\exists z : |S|.\, S^\flat[z]) \supset P(\vec{z}, \vec{y}),$$

where the body $\exists z : |S|.\, S^\flat[z]$ is constructed by quantifier elimination (Definition 6.17) from the family of formulas

$$S^\flat[g] \quad := \quad (g : S)^\flat \wedge p\, \vec{x}\, g\, (z_1, \ldots, z_k)$$

indexed in the quasi-closed normal form terms $g : |S|$, in which the subformula $(g : S)^\flat$ is obtained by index projection (Definition 6.14) and the constraint formula $p\, \vec{x}\, g\, (z_1, \ldots, z_k)$ by $\rightarrow_{\beta c \gamma}$ reduction. Our definition includes $(q : \bigcirc P(\vec{y}))^\flat$ as a special case: for quasi-closed $q : |\bigcirc P(\vec{y})| = \mathbf{U}^k \Rightarrow \mathbf{C}$ we put

$$(q : \bigcirc P(\vec{y}))^\flat \quad := \quad \forall \vec{z}.\, q\, (z_1, \ldots, z_k) \supset P(\vec{z}, \vec{y}),$$

where the formula $q\, (z_1, \ldots, z_k)$ is determined by $\rightarrow_{\beta c \gamma}$ reduction.

The refinement of an abstract program $\Pi = \zeta_1, \ldots, \zeta_n$ by a constraint table Ξ is defined clause-wise: $(\Xi : \Pi)^\flat = (p_1 : \zeta_1)^\flat, \ldots, (p_n : \zeta_n)^\flat$.

Proposition 6.22 *The refined formula $(p : \zeta)^\flat$ determined by Definition 6.21 is a CLP clause. Hence, if Π is an abstract program and Ξ a constraint table for Π, then $(\Xi : \Pi)^\flat$ is a CLP-program.*

Remark: We can generalise the refinement by stipulating $(p : \bigcirc S)^\flat := \forall z : |S|.\, p\, z \supset (z : S)^\flat$, $(p : \forall x.\, M)^\flat := \forall x.\, (p\, x : M)^\flat$, and $(p : S \supset N)^\flat := \forall z : |S|.\, (z : S)^\flat \supset (p\, z : N)^\flat$, where the universal quantifier $\forall z : |S|$ is (defined and) eliminated in an analogous fashion as here $\exists z : |S|$. In this way we may generalise the refinement to clauses of the form $\forall \vec{x}.\, S \supset \bigcirc T$ where S, T are Σ-formulas.

The next result says that the separation $\theta \mapsto \theta^{\sharp_1} : \theta^{\sharp_2}$ of a CLP-clause into an abstract clause $\theta^{\sharp_2}$ and an associated constraint table $\theta^{\sharp_1}$, and the refinement $p : \zeta \mapsto (p : \zeta)^\flat$ are mutually inverse operations, up to logic equivalence. This shows that by separating out the constraints from a CLP-program and making them part of its proof we have not lost any information.

Proposition 6.23 *For any CLP-clause $\theta = \forall \vec{x}.\, S \supset A$, $(\theta^\sharp)^\flat$ is logically equivalent to θ under the usual equality rules of predicate logic, e.g. in QLL(=).*

Remark: Note that here we do not include equality reasoning in LLP. The equivalence between θ and $(\theta^\sharp)^\flat$ stated in Proposition 6.23 is not provable in the calculi $\vdash_{\mathrm{r}}$ or $\vdash_{\mathrm{l}}$. This does not however impinge on the fact that θ and $(\theta^\sharp)^\flat$ have the same behaviour, *i.e.* all answer constraints are identical (up to $\sim$). Thus, θ and $(\theta^\sharp)^\flat$ are interchangeable also w.r.t. $\vdash_{\mathrm{r}}$ and $\vdash_{\mathrm{l}}$.

Example 6.24 *Consider the constraint table $\Theta^{\sharp_1} = \theta_1^{\sharp_1}, \theta_2^{\sharp_1}, \theta_3^{\sharp_1}$ and abstracted program $\Theta^{\sharp_2} = \theta_1^{\sharp_2}, \theta_2^{\sharp_2}, \theta_3^{\sharp_2}$ of Example 6.7. Recombining constraint table and abstraction for the first clause yields*

$$(\theta_1^{\sharp_1} : \theta_1^{\sharp_2})^\flat \quad = \quad \forall s.\, (\exists z : |\mathit{true}|.\, (z : \mathit{true})^\flat \wedge \theta_1^{\sharp_1}\, z\, s) \supset A_1(s).$$

Eliminating the quantifier $\exists z : |\mathit{true}|$ this reduces to

$$(\theta_1^{\sharp_1} : \theta_1^{\sharp_2})^\flat \quad = \quad \forall s.\, (* : \mathit{true})^\flat \wedge \theta_1^{\sharp_1}\, *\, s) \supset A_1(s).$$

From here we use index projection to replace $(* : \mathit{true})^\flat$ by true and $\rightarrow_{\beta c \gamma}$ reduce $\theta_1^{\sharp_1}\, *\, s = (\lambda z.\, \lambda v.\, v \geq 5)\, *\, s$ to $s \geq 5$. Thus, we get

$$(\theta_1^{\sharp_1} : \theta_1^{\sharp_2})^\flat \quad = \quad \forall s.\, (\mathit{true} \wedge s \geq 5) \supset A_1(s),$$

which indeed is equivalent to θ_1. Similarly we find that $(\theta_2^{\sharp_1} : \theta_2^{\sharp_2})^\flat = \forall s.\, (\mathit{true} \wedge s \geq 9) \supset A_2(s)$. For the third clause we proceed as follows. The refinement is

$$(\theta_3^{\sharp_1} : \theta_3^{\sharp_2})^\flat \quad = \quad \forall t.\, (\exists z : |A_1 \wedge A_2 \wedge \mathit{true}|.\, (z : A_1 \wedge A_2 \wedge \mathit{true})^\flat \wedge \theta_3^{\sharp_1}\, z\, t) \supset B(t)$$

by Definition 6.21. Again, by quantifier elimination this is the same as

$$\forall t.\, (\exists z_1, z_2.\, ((z_1, z_2, *) : A_1 \wedge A_2 \wedge \mathit{true})^\flat \wedge \theta_3^{\sharp_1}\, (z_1, z_2, *)\, t) \supset B(t).$$

The index projection $((z_1, z_2, *) : A_1 \wedge A_2 \wedge \mathit{true})^\flat$ gives the formula $A_1(z_1) \wedge A_2(z_2) \wedge \mathit{true}$. As $\theta_3^{\sharp_1}$ is $\lambda z.\, \lambda v.\, \exists s.\, \pi_1 z = s \wedge \pi_2 z = s \wedge v \geq s + 35$, the constraint term $\theta_3^{\sharp_1}\, (z_1, z_2, *)\, t$ reduces to $\exists s.\, z_1 = s \wedge z_2 = s \wedge t \geq s + 35$. Hence,

$$(\theta_3^{\sharp_1} : \theta_3^{\sharp_2})^\flat \quad = \quad \forall t.\, (\exists z_1, z_2.\, A_1(z_1) \wedge A_2(z_2) \wedge \mathit{true} \wedge (\exists s.\, z_1 = s \wedge z_2 = s \wedge t \geq s + 35)) \supset B(t)$$

which is equivalent to θ_3, up to equality reasoning.

Roughly speaking, the difference between θ and $\theta^* = (\theta^{\sharp_1} : \theta^{\sharp_2})^\flat$ is that (i) the body of θ^* is transformed into a disjunctive normal form, *e.g.* $(A_1 \vee A_2) \wedge A_3$ in θ becomes $(A_1 \wedge A_3) \vee (A_2 \wedge A_3)$ in θ^*, and (ii) the constraint parameters of atomic program formulas are broken out, *e.g.* $R(t_1, \ldots, t_n)$ with $\gamma(R) = k$ in θ is replaced by $\exists \vec{z}.\, R(z_1, \ldots, z_k, t_{k+1}, \ldots, t_n) \wedge z_1 = t_1 \wedge \cdots \wedge z_k = t_k$ θ^*. In a sense the mapping $\theta \mapsto \theta^*$ can be viewed as a normalisation process.

Example 6.25 *Take the constraint table and abstracted program for* `mortgage` *from Example 6.8:*

$$\begin{array}{rcl} \mathtt{M}^{\sharp_2} & = & \forall \mathtt{D}.\, (\mathtt{D} \leq 1 \wedge \mathit{true}) \;\vee\; (\mathtt{D} > 1 \wedge \mathtt{m}(\mathtt{D} - 1)) \;\supset\; \bigcirc \mathtt{m}(\mathtt{D}) \\ \mathtt{M}^{\sharp_1} & = & \lambda \mathtt{D}, \mathtt{z}, \mathtt{v}.\, \mathsf{case}\ \mathtt{z}\ \mathsf{of}\ [\iota_1(\mathtt{z}_1) \rightarrow \mathtt{c}_1(\mathtt{v}), \iota_2(\mathtt{z}_2) \rightarrow \mathtt{c}_2(\mathtt{v}, \mathtt{z}_2)]. \end{array}$$

Going through the definition of refinement, with quantifier elimination, index projection and $\rightarrow_{\beta c \gamma}$ reduction yields

$$\begin{array}{l} (\mathtt{M}^{\sharp_1} : \mathtt{M}^{\sharp_2})^\flat \;= \\ \quad \forall \mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, \mathtt{D}.\, ((\mathtt{D} \leq 1 \wedge \mathit{true} \wedge \mathit{true} \wedge \mathtt{B} + \mathtt{MP} = \mathtt{P} * (\mathtt{I} + 1)) \vee \\ \qquad\qquad (\exists \mathtt{z}_1, \mathtt{z}_2, \mathtt{z}_3, \mathtt{z}_4.\, \mathtt{D} > 1 \wedge \mathtt{m}(\mathtt{z}_1, \mathtt{z}_2, \mathtt{z}_3, \mathtt{z}_4, \mathtt{D} - 1) \wedge \mathtt{z}_1 = \mathtt{P} * (\mathtt{I} + 1) - \mathtt{MP} \wedge \\ \qquad\qquad\qquad \mathtt{z}_2 = \mathtt{I} \wedge \mathtt{z}_3 = \mathtt{MP} \wedge \mathtt{z}_4 = \mathtt{B})) \;\supset \mathtt{m}(\mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, \mathtt{D}). \end{array}$$

which is equivalent to clause `M` *of Example 2.1, up to equality reasoning.*

The refinement on programs, too, is mirrored by a refinement of proofs.

Theorem 6.26 (Soundness of Refinement) *Let $\Pi = \zeta_1, \ldots, \zeta_n$ be an abstract LLP-program, $\Xi = p_1, \ldots, p_n$ a constraint table for Π, and A an atomic query, such that*

$$z_1 : \zeta_1, \ldots, z_n : \zeta_n \vdash_{\mathrm{LLP}} q : \bigcirc A$$

for some proof term q. Then,

$$(p_1 : \zeta_1)^\flat, \ldots, (p_n : \zeta_n)^\flat \vdash_{\mathrm{QLL}} (|q\{p_1, \ldots, p_n / z_1, \ldots, z_n\}| : \bigcirc A)^\flat$$

Theorem 6.26 is our main theorem. It basically means that we can use the calculus of LLP ($\vdash_{\mathrm{r}}$ or $\vdash_{\mathrm{l}}$) as a two-phase CLP system with clean separation between constraint generation and constraint analysis: Given a CLP-program $\Theta = \theta_1, \ldots, \theta_n$ and a query[12] $P(y_1, \ldots, y_m)$ we first run the abstracted program $\Theta^{\sharp_2} = \theta_1^{\sharp_2}, \ldots, \theta_n^{\sharp_2}$ on the abstracted query $\bigcirc P(y_{k+1}, \ldots, y_m)$, assuming $k = \gamma(P)$, which, say, yields a proof q such that

$$z_1 : \theta_1^{\sharp_2}, \ldots, z_n : \theta_n^{\sharp_2} \quad \vdash_{\mathrm{LLP}} \quad q : \bigcirc P(y_{k+1}, \ldots, y_m). \tag{7}$$

This proof or execution is controlled by the abstract structure of the program, and establishes a particular solution q for the query. From the proof term q we can now extract the answer constraint $|q|$ (a $\lambda_c(\mathcal{C})$-term of type $|\bigcirc P(y_{k+1}, \ldots, y_m)| = \mathbf{U}^k \Rightarrow \mathbf{C}$) for the chosen abstract execution. We simply substitute the constraint tables $\theta_i^{\sharp_1}$ for the proof variables z_i of $|q|$ obtaining a quasi-closed $\lambda_c(\mathcal{C})$-term $q^* = |q\{\theta_i^{\sharp_1}/z_i \mid i = 1, \ldots, n\}|$ of type $\mathbf{U}^k \Rightarrow \mathbf{C}$. From the Soundness Theorem 6.26 for refinement, then, we conclude that

$$\theta_1^*, \ldots, \theta_n^* \quad \vdash_{\mathrm{QLL}} \quad \forall y_1, \ldots, y_k.\, q^*\,(y_1, \ldots, y_k) \supset P(y_1, \ldots, y_m), \tag{8}$$

where $\theta_i^* = \theta_i^{\sharp^\flat}$. So, what have we achieved? We started off with an abstract query $P(y_{k+1}, \ldots, y_m)$ and an abstract proof (7) of this query from the abstracted program $\Theta^{\sharp_2}$, and ended up with an answer constraint $q^*\,(y_1, \ldots, y_k)$. The concrete-level proof (8) verifies that the answer constraint is sound. That this two-phase CLP-system is complete w.r.t. to conventional CLP will be shown in Section 7.

It should be stressed that, operationally speaking, the answer constraint indeed has been computed in the true sense of the word, *i.e.* it has been synthesised. More specifically, the computation is performed in two phases: in phase one we construct an abstract proof q, using the rules of the LLP-calculus as an operational semantics, and in phase two we reduce the term $q^*\,(y_1, \ldots, y_k)$ using the computational semantics of the $\lambda_c(\mathcal{C})$-calculus. This contrasts with the traditional methods of giving a logical account of CLP which merely provide a "verification" type, *i.e.* provability, semantics for CLP: The logic rules of ordinary sequent calculi cannot be used to construct[13] the answer constraint. They can only be used *post hoc*, *i.e.* for proving that a given constraint formula really is a valid constraint for a particular query w.r.t. a particular program (see *e.g.* [HSH90]). The main feature of LLP, not present in standard logics, that makes this operational interpretation possible is the constraint modality $\bigcirc$. Another important aspect that becomes visible at this point

[12] We can also include constraints into the query using a "double implication trick." For instance, to run the query `I = 0.01, B = 0, mortgage(P,I,MP,B,5)` we extend the CLP-program `mortgage` by an additional clause $\forall \mathtt{P}, \mathtt{MP}. (\exists \mathtt{I}, \mathtt{B}.\, \mathtt{I} = 0.01 \wedge \mathtt{B} = 0 \wedge \mathtt{mortgage}(\mathtt{P}, \mathtt{I}, \mathtt{MP}, \mathtt{B}, 5)) \supset \mathtt{result}(\mathtt{P}, \mathtt{MP})$ with a new predicate symbol `result`, and then run the new query `result(P,MP)` on the extended program.

[13] This requires extra-logical features such as meta-variables and higher-order sequents as *e.g.* in the LAMBDA or ISABELLE theorem provers.

is that our constraint generation method is more general than that of standard CLP systems since the resulting constraint $q^*\,(y_1, \ldots, y_k)$ may introduce implicit variables for the query. In traditional CLP systems the constraint obtained from running a query $P(y_{k+1}, \ldots, y_m)$ is a condition on the explicit variables $y_{k+1}, \ldots, y_m$ only. As we will see from our running example to be discussed below, this extra generality is very useful. It means that the answer constraint in general may also produce a specialisation of the original query and provide additional intensional information that has not been anticipated at the abstract level already. Such unobservable parameters might, for instance, relate to debugging or profiling information, or user input.

Example 6.27 *To finish off this section let us take up our running Example 6.1 again. Suppose we are interested primarily in the functional (here: synchronisation) behaviour of Θ, and thus temporarily move the timing details out of the way. Thus, we apply constraint abstraction to produce the purely functional behaviour $\Theta^{\sharp_2} = \theta_1^{\sharp_2}, \theta_2^{\sharp_2}, \theta_3^{\sharp_2}$ of the program and a separate constraint table $\Theta^{\sharp_1} = \theta_1^{\sharp_1}, \theta_2^{\sharp_1}, \theta_3^{\sharp_1}$ that contains all information about the timing. These two parts have been determined already in Example 6.7 to be*

$$\theta_1^{\sharp_2} = \mathit{true} \supset \bigcirc A_1 \qquad \theta_2^{\sharp_2} = \mathit{true} \supset \bigcirc A_2 \qquad \theta_3^{\sharp_2} = (A_1 \wedge A_2 \wedge \mathit{true}) \supset \bigcirc B$$

and

$$\theta_1^{\sharp_1} = \lambda z.\, \lambda v.\, v \geq 5 \quad \theta_2^{\sharp_1} = \lambda z.\, \lambda v.\, v \geq 9 \quad \theta_3^{\sharp_1} = \lambda z.\, \lambda v.\, \exists s.\, \pi_1 z = s \wedge \pi_2 z = s \wedge v \geq s + 35.$$

It is now possible to verify the abstract program with respect to a query, say $\bigcirc B$, i.e. prove the sequent $\Theta^{\sharp_2} \vdash_{\mathrm{LLP}} \bigcirc B$. The modality $\bigcirc$ indicates that we expect the query B to verify up to some timing constraint. Fig. 11 shows such a LLP proof using $\vdash_{\mathrm{r}}$ as in Example 6.10, where $\Pi = x_1 : \theta_1^{\sharp_2}, x_2 : \theta_2^{\sharp_1}, x_3 : \theta_3^{\sharp_2}$. The generic meta-variable $q^\uparrow$ denotes the proof term of the assertion in the sequent directly above; $q_L^\uparrow$, $q_R^\uparrow$ abbreviate the proof terms of the left and right sequents' assertions, respectively. For the sake of clarity, the subscript is omitted from the entailment symbols. The

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \mathit{true}_\bigcirc : \bigcirc \mathit{true}}\,\mathit{true}_\bigcirc}{\Pi \vdash \supset_\bigcirc(q^\uparrow, x_1) : \bigcirc A_1}\,\supset_\bigcirc \quad \dfrac{\dfrac{}{\Pi \vdash \mathit{true}_\bigcirc : \bigcirc \mathit{true}}\,\mathit{true}_\bigcirc}{\Pi \vdash \supset_\bigcirc(q^\uparrow, x_2) : \bigcirc A_2}\,\supset_\bigcirc}{\Pi \vdash \wedge_\bigcirc(q_L^\uparrow, q_R^\uparrow) : \bigcirc(A_1 \wedge A_2)}\,\wedge_\bigcirc \quad \dfrac{}{\Pi \vdash \mathit{true}_\bigcirc : \bigcirc \mathit{true}}\,\mathit{true}_\bigcirc}{\dfrac{\Pi \vdash \wedge_\bigcirc(q_L^\uparrow, q_R^\uparrow) : \bigcirc(A_1 \wedge A_2 \wedge \mathit{true})}{\Pi \vdash \supset_\bigcirc(q^\uparrow, x_3) : \bigcirc B}\,\supset_\bigcirc}\,\wedge_\bigcirc$$

Figure 11: A Simple Abstract Proof

abstract proof builds up a proof term $q = \supset_\bigcirc(q^\uparrow, x_3)$ such that $\Pi \vdash_{\mathrm{r}} q : \bigcirc B$, which expands to

$$q = \supset_\bigcirc(\wedge_\bigcirc(\wedge_\bigcirc(\supset_\bigcirc(\mathit{true}_\bigcirc, x_1), \supset_\bigcirc(\mathit{true}_\bigcirc, x_2)), \mathit{true}_\bigcirc), x_3)$$

thus verifying that the abstract system $\Theta^{\sharp_2}$ produces a functional output B. Term q contains the constraint information, i.e. at what time this output appears. We first unroll the derived rules as proof terms in the $\lambda_c^{\Sigma\Pi}$ calculus and get

$$\begin{array}{rl} q = & \mathsf{let}\ z_1 \Leftarrow \mathsf{let}\ y_1 \Leftarrow \mathsf{let}\ y_2 \Leftarrow \mathsf{let}\ z_4 \Leftarrow \mathsf{val}(*) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ x_1\, z_4 \\ & \qquad\qquad\qquad\qquad\quad\ \mathsf{in}\ \ \mathsf{let}\ z_3 \Leftarrow \mathsf{let}\ z_5 \Leftarrow \mathsf{val}(*) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ x_2\, z_5 \\ & \qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{val}(y_2, z_3) \\ & \qquad\qquad\quad \mathsf{in}\ \ \mathsf{let}\ z_2 \Leftarrow \mathsf{val}(*) \\ & \qquad\qquad\qquad \mathsf{in}\ \mathsf{val}(y_1, z_2) \\ & \mathsf{in}\ x_3\, z_1 \end{array}$$

which, in a purely structural sense, represents the bare bones of the proof tree in Fig. 11. Yet, the $\lambda_c^{\Sigma\Pi}$-term q is not just structure but comes equipped with a semantics, too. Indeed, the $\bigcirc.\beta$ equation of the $\lambda_c^{\Sigma\Pi}$ calculus (see Fig. 4), which we require to hold for every notion of constraint, allows us to eliminate the three occurrences of $\mathsf{val}(*)$ *immediately and simplify the term as follows:*

$$\begin{aligned} q \;=\; & \mathsf{let}\ z_1 \Leftarrow \mathsf{let}\ y_1 \Leftarrow \mathsf{let}\ y_2 \Leftarrow x_1 * \\ & \qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{let}\ z_3 \Leftarrow x_2 * \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{val}(y_2, z_3) \\ & \qquad\qquad\qquad \mathsf{in}\ \mathsf{val}(y_1, *) \\ & \mathsf{in}\ x_3 z_1 . \end{aligned}$$

The remaining $\mathsf{val}(y_2, z_3)$, $\mathsf{val}(y_1, *)$ *admit further $\bigcirc.\beta$-reductions, however we must first rearrange the* let-in *structure suitably by several applications of $\bigcirc.ass$-reduction:*

$$\begin{aligned} q \;=\; & \mathsf{let}\ y_2 \Leftarrow x_1 * \\ & \quad \mathsf{in}\ \ \mathsf{let}\ z_3 \Leftarrow x_2 * \\ & \qquad \mathsf{in}\ \ \mathsf{let}\ y_1 \Leftarrow \mathsf{val}(y_2, z_3) \\ & \qquad\quad \mathsf{in}\ \ \mathsf{let}\ z_1 \Leftarrow \mathsf{val}(y_1, *) \\ & \qquad\qquad \mathsf{in}\ x_3\, z_1 . \end{aligned}$$

Now we can apply $\bigcirc.\beta$-reduction again and obtain

$$q \quad = \quad \mathsf{let}\ y_2 \Leftarrow x_1\ *\ \mathsf{in}\ \mathsf{let}\ z_3 \Leftarrow x_2\ *\ \mathsf{in}\ x_3\,((y_2, z_3), *).$$

At this point we do not get any further with $\lambda_c^{\Sigma\Pi}$ and the generic notion of constraint. The proof term q, now, is in $\rightarrow_{\beta c}$ normal form. If we had used the system $\vdash_{\mathrm{l}}$ we would have got the proof term $q' =$ $\mathsf{let}\ z \Leftarrow q\ \mathsf{in}\ \mathsf{val}(z)$ *right away, which is only one $\bigcirc.\eta$ equation away from the normal form q. We have used the $\vdash_{\mathrm{r}}$ system here to demonstrate that in general sub-structural QLL calculi for LLP require $\rightarrow_{\beta c}$ reductions to compute constraints. To carry on with constraint evaluation, we now use the interpretation mapping $|\cdot|$ to translate into the $\lambda_c(\mathcal{C})$-calculus, where we can substitute the concrete constraint tables $\theta_1^{\sharp_1}$, $\theta_2^{\sharp_1}$, $\theta_3^{\sharp_1}$ for proof variables x_1, x_2, x_3, respectively:*

$$\begin{aligned} q^* \;:=\; & |q|\{\theta_1^{\sharp_1}, \theta_2^{\sharp_1}, \theta_3^{\sharp_1} / x_1, x_2, x_3\} \\ =\; & \mathsf{let}\ y_2 \Leftarrow \theta_1^{\sharp_1}\ *\ \mathsf{in}\ \mathsf{let}\ z_3 \Leftarrow \theta_2^{\sharp_1}\ *\ \mathsf{in}\ \theta_3^{\sharp_1}\,((y_2, z_3), *). \end{aligned}$$

This term has type $|\bigcirc B| = \mathbf{U} \Rightarrow \mathbf{C}$. The timing constraint we are after is $q^*\, t$ which we compute by applying the equations $\Rightarrow.\beta$, $\mathbf{C}.\mathsf{let}$ *of $\lambda_c(\mathcal{C})$:*

$$\begin{aligned} q^*\, t \;&=\; (\mathsf{let}\ y_2 \Leftarrow \theta_1^{\sharp_1}\ *\ \mathsf{in}\ \mathsf{let}\ z_3 \Leftarrow \theta_2^{\sharp_1}\ *\ \mathsf{in}\ \theta_3^{\sharp_1}\,((y_2, z_3), *))\, t \\ &=\; \exists y_2.\, \theta_1^{\sharp_1}\ *\ y_2\ \wedge\ (\mathsf{let}\ z_3 \Leftarrow \theta_2^{\sharp_1}\ *\ \mathsf{in}\ \theta_3^{\sharp_1}\,((y_2, z_3), *))\, t \\ &=\; \exists y_2.\, y_2 \geq 5\ \wedge\ \exists z_3.\, \theta_2^{\sharp_1}\ *\ z_3\ \wedge\ \theta_3^{\sharp_1}\,((y_2, z_3), *)\, t \\ &=\; \exists y_2.\, y_2 \geq 5\ \wedge\ (\exists z_3.\, z_3 \geq 9\ \wedge\ (\exists s.\, y_2 = s \wedge z_3 = s \wedge t \geq s + 35)). \end{aligned}$$

Note that we are entitled to apply the equation $\mathbf{C}.\mathsf{let}$ *since the expressions are of the right elementary types, i.e. in particular that y_2, z_3 are of type $\mathbf{U}$. Having got this far, a simple constraint solver for one-sided inequations would reduce this expression for $q^*\, t$ to the equivalent constraint $t \geq 44$. Theorem 6.26 and Proposition 6.23 now imply that in fact $\Theta \vdash_{\mathrm{QLL}} t \geq 44 \supset B(t)$ in a QLL theory QLL($\mathcal{C}$) that includes equality reasoning and constraint equivalence, so that QLL($\mathcal{C}$) $\vdash c \equiv d$ whenever $c \sim d$.*

The example demonstrates how the essence of constraint logic programming is nicely captured in our logic framework. Instead of manipulating a flat unstructured clause like $\forall s.\ s \geq 5 \supset A_1(s)$ it

separates the built-in constraint predicates and the user programs to give pairs, like $\lambda z.\,\lambda v.\, v \geq 5 : \mathit{true} \supset \bigcirc A_1$, and works at the level of such pairs. This separation of concerns is at the heart of CLP but rarely made explicit in CLP semantics based on logic (see, *e.g.* [DGW91]). Instead of running Θ at the concrete level with function and timing intertwined we have abstracted from timing, verified a purely functional query, and as a side-product obtained a timing constraint $t \geq 44$ by proof extraction. The reader should observe that all the constructions involved in extracting the timing constraint were done by $\rightarrow_{\beta c\gamma}$ reduction only, whence they can be automated. The computational bit missing here is the constraint solver, which we consider part of $\lambda_c(\mathcal{C})$-reduction (extending $\rightarrow_{\beta c\gamma}$ reduction) in a practical implementation of the method as a CLP system. A very similar, albeit more specialised, version of timing abstraction is proposed in [Men96, MF96] as a framework for extracting timing information for combinational circuits.
The example can well be run with the proper functionality of first-order terms included as in the original formulation of Example 6.1. The reader may find it instructive to fill in the first-order terms from Example 6.1. Then, the fundamental difference between constraint *synthesis*, which proceeds at the level of proofs, and functional *verification*, which is done at the level of formulas, should become apparent.

The next, and final, example of this section shows that sometimes it can be more advantageous to compute constraints at the abstract level of $\lambda_c^{\Sigma\Pi}$ proof terms than at the concrete level of constraint propositions. This supports our proposal that constraints should be distinguished from propositions. It also shows that our method computationally is more powerful than the solutions plans of [DGW91], for computations in $\lambda_c^{\Sigma\Pi}$ can perform actual substitutions, *i.e.* have a functional flavour, while the semantics of solutions plans only produces equations, *i.e.* is of relational nature (see our discussions on related work in Section 9.2).

Example 6.28 *Suppose R is a relation symbol with $\gamma(R) = 1$. Then $|\bigcirc R(t)| = \mathbf{U} \Rightarrow \mathbf{C}$. A constraint table for $\bigcirc R(t)$ is $\mathsf{val}(s) : \mathbf{U} \Rightarrow \mathbf{C}$ for a arbitrary object-level term s. If we refine $p_1 : \zeta_1 := \mathsf{val}(s) : \bigcirc R(t)$ to the concrete level immediately, we get $(\mathsf{val}(s) : \bigcirc R(t))^\flat = \forall x.\, \mathsf{val}(s)\, x \supset R(x,t)$ which after $\lambda_c(\mathcal{C})$-reduction gives $\theta_1 := \forall x.\, s = x \supset R(x,t)$. This is logically equivalent to $\theta_1^* := R(s,t)$ but only by reference to additional equational reasoning. The abstract pair $\mathsf{val}(s) : \bigcirc R(t)$ however does behave like $R(s,t)$ by virtue of the computational semantics of $\lambda_c^{\Sigma\Pi}$. To see this, consider another[14] abstract clause $p_2 : \zeta_2 := \lambda y, x.\mathsf{val}(r) : \forall y.\, R(y) \supset \bigcirc P(y)$, where $\gamma(P) = 1$. This clause refines to $(p_2 : \zeta_2)^\flat = \forall z, y.\, (\exists x.\, R(x,y) \wedge (\lambda y, x.\mathsf{val}(r))\, y\, x\, z) \supset P(z,y)$ which in $\lambda_c(\mathcal{C})$ reduces to*

$$\theta_2 \quad := \quad \forall z, y.\, (\exists x.\, R(x,y) \wedge r = z) \supset P(z,y).$$

Again this can be simplified to $\theta_2^* := \forall y, x.\, R(x,y) \supset P(r,y)$ but this involves equational reasoning. If we "execute" the abstract clauses $p_1 : \zeta_1$ and $p_2 : \zeta_2$ we can construct a proof

$$\mathsf{let}\ v \Leftarrow \mathsf{val}(s)\ \mathsf{in}\ (\lambda y, x.\mathsf{val}(r))\, t\, v \quad : \quad \bigcirc P(t),$$

which refines to

$$\forall z.\, (\mathsf{let}\ v \Leftarrow \mathsf{val}(s)\ \mathsf{in}\ (\lambda y, x.\mathsf{val}(r))\, t)\, z \supset P(z,t),$$

and finally in $\lambda_c(\mathcal{C})$ reduces to $\forall z.\, r\{t/y\}\{s/x\} = z \supset P(z,t)$. Thus, we have performed a real substitution of s for x, which is what we wanted and what we should expect from composing the intended clauses θ_1^*, θ_2^*. If, however, we had composed the concrete level refinements θ_1 and θ_2 as CLP programs then we would end up with $\forall x, z.\, s = x \wedge t = r\{t/y\} \supset P(z,t)$ in which the substitution is only implicit in the equation $s = x$. This shows that the computational semantics of $\lambda_c^{\Sigma\Pi}$ does perform some amount of constraint computation without resorting to an equational constraint solver.

[14]Strictly speaking, $\bigcirc R(t)$ is not exactly a clause according to our definition, but this does not make a difference, since the argument works also with the proper clause $\forall z.\, z = t \supset \bigcirc R(z)$.

7 Embedding CLP proofs into LLP

Various operational semantics for general CLP schemes have been proposed which use a notion of resolution, or derivation rules (see for example [AV96, GDL95, And92] or the survey [JM94]). This section illustrates one such (top-down) procedural semantics, as proposed in [DG94] or [AV96][15], closely related to our $\vdash_1$, and proves that the answer constraints generated by them are equivalent to the constraints extracted from an abstract proof of the same program in LLP. More specifically, the abstraction is the special case without hidden parameters, *i.e.* $\gamma(R) = 0$ for all relation symbols R.

The state of a CLP system is represented by *goals* and the resolution process is a sequence of steps which lead from one goal to another. We assume a fixed notion of constraint $\mathcal{C} = (\Phi, \sim)$ with trivially solvable constraint $true \in \Phi$. A constraint $c \in \Phi$ is *solvable* if the existential closure $\exists\vec{x}.\,c$ of c satisfies $\exists\vec{x}.\,c \sim true$.

Definition 7.1 (Goal) *A* goal $\mathcal{G}$ *has the form* $c \,\Box\, S_1, \ldots, S_n$*, where* $c \in \Phi$ *is a constraint and* S_i $(i \in \{1, \ldots, n\})$ *are* Σ*-formulas.*

Definition 7.2 (Derivation Step) *A* derivation step *for a goal of the form* $\mathcal{G} = c \,\Box\, S_1, \ldots, S_n$ *in the program* Π *results in a goal of the form* $\mathcal{G}' = c' \,\Box\, S_1, \ldots, S_{i-1}, \Gamma, S_{i+1}, \ldots, S_n$*, written*

$$c \,\Box\, S_1, \ldots, S_n \overset{\Pi}{\underset{1}{\leadsto}} c' \,\Box\, S_1, \ldots, \Gamma, \ldots, S_n$$

where Γ *is a list of* Σ*-formulas, constraint* c' *is solvable, and* S_i*,* Γ *and* c' *are determined by Table 1. The number* 1 *beneath the* $\leadsto$ *symbol identifies a single derivation step.*

Rule	S_i	Γ	c'
0	$true$		c
1	B		$c \wedge B$
2a	$T_1 \vee T_2$	T_1	c
2b	$T_1 \vee T_2$	T_2	c
3	$T_1 \wedge T_2$	T_1, T_2	c
4	$\exists x.\,T$	$T\{u/x\}$	c
5	$P(\vec{t})$	$T\{\vec{t}/\vec{y}\}\{\vec{u}/\vec{x}\}$	c

Note: in rule 1, B is a constraint; in rule 4, the variable u does not occur in $\mathcal{G}$; and for rule 5, $w : \forall\vec{x}.\,T \supset P(\vec{y})$, with $\vec{x} = x_1, \ldots x_m$, $\vec{y} = x_1, \ldots, x_n$, $n \le m$, is a clause in Π, and moreover variables $\vec{u} = u_{n+1}, \ldots, u_m$ do not occur in $\mathcal{G}$.

Table 1: Goal reduction for derivation steps

Definition 7.3 (Derivation) *A* derivation *of a goal* $\mathcal{G}$ *in a program* Π *is a finite or infinite sequence of goals such that every goal, apart from* $\mathcal{G}$*, is obtained from the previous one by means of a derivation step in* Π*. A* successful derivation *of a goal* $\mathcal{G}$ *is a finite sequence whose last element is a goal of the form* $c' \,\Box\, \varepsilon$ *(i.e. no more program clauses left to consider). In this case,* c' *is called the* answer constraint*.*

[15]The paper indicated also discusses an extension to CLP in the form of *bounded quantifiers*. For the sake of brevity, such extensions are not considered here.

It may be pointed out that this operational semantics slightly deviates from [AV96] in that our derivations do not introduce any existential quantification in the generation of constraints. On the face of it this may seem a severe restriction since existential quantifiers are an essential tool to localise variables. However, this is justified since, despite their appearance, they do not play a very important rôle in [AV96]. In fact, constraints in [AV96] are assumed to be in prenex normal form $\exists z.\, c$ where c is a conjunction of atomic constraints, and this normal form is maintained in derivations, like

$$\exists \vec{z}.\, c \,\square\, \vec{S} \overset{\Pi}{\underset{*}{\rightsquigarrow}} \exists \vec{z}.\, \exists \vec{u}.\, c \wedge c' \,\square\, \vec{S}',$$

where $\exists \vec{u}$ binds a number of additional variables introduced in the course of the derivation. Since these extra variables can always be determined and quantified *after* the derivation has been done, there is no need to carry around the quantifiers *during* the derivation. Thus, we may as well drop them completely, as we do here, or as done in the operational semantics for *constrained SLD resolution* given by [DG94]. This contrasts with the proposed LLP approach of constraint generation by proof extraction, which does return truly localised constraint variables as we have seen, *e.g.* in Example 6.27. In our LLP framework the standard notion of derivation as formalised in Table 1 is a special case in which no constraint parameters are hidden. This is made precise by the following two theorems.

Theorem 7.4 (Completeness) *Let $\Theta = \theta_1, \ldots, \theta_k$ be a CLP program and*

$$true \,\square\, A \overset{\Theta}{\underset{*}{\rightsquigarrow}} true \wedge c \,\square\, \varepsilon$$

be a derivation with atomic query A. Then, there exists an abstract proof

$$x_1 : \theta_1^{\sharp_2}, \ldots, x_k : \theta_k^{\sharp_2} \vdash_{\mathrm{LLP}} q : \bigcirc A$$

such that $|q\{x_1, \ldots, x_k / \theta_1^{\sharp_1}, \ldots, \theta_k^{\sharp_1}\}|{*}$ is identical to c up to $\lambda_c(\mathcal{C})$-reductions and reordering of conjunctions, assuming that no constraint parameters are hidden, i.e. $\gamma(R) = 0$ for all relation symbols R. Note, this implies $A^{\sharp_2} = A$.

Theorem 7.5 (Soundness) *Let $\Theta = \theta_1, \ldots, \theta_k$ be a CLP program and $\vec{w} : \Theta^{\sharp_2} \vdash_{\mathrm{LLP}} p : \bigcirc A$ a proof of a modalised atomic query $\bigcirc A$ from the abstracted program $\Theta^{\sharp_2}$, where $\gamma(R) = 0$ for all relation symbols R. Let $p^* := |p\{\vec{\Theta}^{\sharp_1}/\vec{x}\}|{*}$ be the constraint extracted from p (reduced to normal form) for the constraint table $\Theta^{\sharp_1}$. Then, if p^* is solvable, then there exists a derivation*

$$true \,\square\, A \overset{\Theta}{\underset{*}{\rightsquigarrow}} true \wedge c \,\square\, \varepsilon$$

such that $c \sim p^*$.

Both Theorems 7.5 and 7.4 amount to the statement that abstract verification in LLP, without hidden parameters, in combination with constraint extraction is sound and complete for the standard CLP procedural semantics. In other words, the sequent calculi LLP not only capture the usual notions of provability and truth for CLP but also the operational aspect of constraint generation.

8 Kripke Models for QLL

While the previous sections were focusing heavily on the intensional semantics, this final section explores an extensional model theory for QLL. This rounds off the picture and justifies our calling QLL a modal logic and $\bigcirc$ a modal operator, according to more standard traditions. Also it shows that QLL in a single logic framework combines both model-theoretic and operational aspects of CLP.

The models for QLL, introduced here are a variant of Kripke structures with a single set of worlds, two accessibility relations and fallible worlds. They are called *constraint models* and for propositional logic have been introduced in [FM95]. They are extended here to first-order models in a standard way:

Definition 8.1 *A* Kripke Constraint Model *of* QLL *is a quintuple:*

$$\mathcal{M} = \langle\, W,\, R_i,\, R_m,\, I,\, F\,\rangle$$

where W is a non-empty set of 'possible worlds', R_m and R_i are pre-orders over W such that $R_m \subseteq R_i$, I is a family of first-order interpretations $I = \{I^\alpha \mid \alpha \in W\}$ subject to certain conditions specified below. $F \subseteq W$ is a set of fallible worlds such that $\alpha\ R_i\ \beta$ and $\alpha \in F$ implies $\beta \in F$. With $\mathsf{dom}(\alpha)$ denoting the (nonempty) universe on which I^α is built the following monotonicity *conditions are imposed on I: for all α, β such that $\alpha\ R_i\ \beta$, (i) $\alpha\ R_i\ \beta\ \rightarrow\ \mathsf{dom}(\alpha) \subseteq \mathsf{dom}(\beta)$, (ii) $I^\alpha(c) = I^\beta(c)$ for constants c, (iii) $I^\alpha(f) \subseteq I^\beta(f)$ for function symbols f, (iv) $I^\alpha(P) \subseteq I^\beta(P)$ for relation symbols P.*

As usual, we have the notion of an assignment

$$\rho : \Pi_{\alpha \in W}\, \mathit{Var} \rightarrow \mathsf{dom}(\alpha)$$

as a W-indexed family of functions mapping variables to elements: the terminology ρ^α is employed to denote the function

$$\rho^\alpha : \mathit{Var} \rightarrow \mathsf{dom}(\alpha).$$

Thus we have an assignment on a model that defines for every variable x and world α an element $\rho^\alpha(x) \in \mathsf{dom}(\alpha)$. We lift assignments ρ to valuations ρ_* of terms in the standard way.

Definition 8.2 *Let $\mathcal{M} = \langle\, W,\, R_i,\, R_m,\, I,\, F\,\rangle$ be a Kripke constraint model for* QLL*. A formula, M is said to be* valid *at a world α in $\mathcal{M}$ and under an assignment ρ^α, written $\mathcal{M}, \alpha \models_{\rho^\alpha} M$, according to following conditions:*

- $\mathcal{M}, \alpha \models_{\rho^\alpha} P_n(t_1, \ldots, t_n) \;\Leftrightarrow\; (\rho^\alpha_*(t_1), \ldots, \rho^\alpha_*(t_n)) \in I^\alpha(P_n)$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} \mathit{false} \;\Leftrightarrow\; \alpha \in F$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} \neg M \;\Leftrightarrow\;$ *for all* β *such that* $\alpha\ R_i\ \beta$, $\mathcal{M}, \beta \not\models_{\rho^\beta} M$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} M \wedge N \;\Leftrightarrow\; \mathcal{M}, \alpha \models_{\rho^\alpha} M$ *and* $\mathcal{M}, \alpha \models_{\rho^\alpha} N$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} M \vee N \;\Leftrightarrow\; \mathcal{M}, \alpha \models_{\rho^\alpha} M$ *or* $\mathcal{M}, \alpha \models_{\rho^\alpha} N$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} M \supset N \;\Leftrightarrow\;$ *for all* β *such that* $\alpha\ R_i\ \beta$, $\mathcal{M}, \beta \models_{\rho^\beta} M$ *implies* $\mathcal{M}, \beta \models_{\rho^\beta} N$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} \bigcirc M \;\Leftrightarrow\;$ *for all* β *s.t.* $\alpha\ R_i\ \beta$ *there exists* γ *s.t.* $\beta\ R_m\ \gamma$, $\mathcal{M}, \gamma \models_{\rho^\gamma} M$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} \exists x\, M \;\Leftrightarrow\; \mathcal{M}, \alpha \models_{[c/x]\rho^\alpha} M$, *for some* $c \in \mathsf{dom}(\alpha)$
- $\mathcal{M}, \alpha \models_{\rho^\alpha} \forall x\, M \;\Leftrightarrow\;$ *for all* β *such that* $\alpha\ R_i\ \beta$, $\mathcal{M}, \beta \models_{[c/x]\rho^\beta} M$, *for all* $c \in \mathsf{dom}(\beta)$

A formula M is valid in $\mathcal{M}$ *under assignment ρ, written $\mathcal{M} \models_\rho M$ if, for all $\alpha \in W$, M is valid at α in $\mathcal{M}$ under ρ^α. A formula M is* valid, *written $\models_{\mathrm{QLL}} M$, if M is valid in any $\mathcal{M}$ for any ρ.*

It is well known that intuitionistic logic can be embedded in the classical modal logic $S4$, using Gödel's translation [Göd69]. By an extension of this translation, we may embed QLL in a classical bi-modal logic which we call $[S4, S4]$. This logic has the usual classical logical connectives together with *two* primitive modalities, $\Box_i$ and $\Box_m$, each of which possesses $S4$ properties. The purpose of multiple modalities is to relate the fundamental intuitionistic nature of QLL to $\Box_i$ and the modality of QLL to $\Box_m$. By first-order extensions of results in [Pop94], $[S4, S4]$ can be shown to be sound and (Kripke) complete for the class of $[S4, S4]$-models.
The following function * translates formulas in the language of QLL into formulas in the language of $[S4, S4]$. Here, f is a distinguished propositional constant, which is *not* the same as '*false*'

$$\begin{aligned} \mathit{false}^* &= \Box_i \mathrm{f} & (M \supset N)^* &= \Box_i (M^* \supset N^*) \\ P^* &= \Box_i P \vee \Box_i \mathrm{f} & (\exists x.M)^* &= \exists x.M^* \\ (M \vee N)^* &= M^* \vee N^* & (\forall x.M)^* &= \Box_i \forall x.M^* \\ (M \wedge N)^* &= M^* \wedge N^* & (\bigcirc M)^* &= \Box_i \neg \Box_m \neg M^* \end{aligned}$$

This translation can be used to prove syntactic and semantic embedding/reflection results between QLL and $[S4, S4]$. In the following, $\vdash_{[\mathrm{S4,S4}]} \varphi$ denotes that the sequent $\vdash \varphi$ is a theorem of $[S4, S4]$, and $\models_{[S4,S4]} \varphi$ denotes that the formula φ is valid in all $[S4, S4]$ models.

Lemma 8.3 (Syntactic Embedding/Reflection) *A formula M is derivable in* QLL *iff the translated formula M^* is a theorem of* $[S4, S4]$*, i.e.*

$$\vdash_{\mathrm{QLL}} M \iff \vdash_{[\mathrm{S4,S4}]} M^*$$

We also have embedding and reflection results for the model theories.

Lemma 8.4 (Semantic Embedding/Reflection) *For all formulas M of* QLL*,*

$$\models_{\mathrm{QLL}} M \iff \models_{[S4,S4]} M^*$$

Therefore, by the soundness and completeness of $[S4, S4]$, we may conclude soundness and completeness for QLL.

Theorem 8.5 (Soundness and Completeness) *For all formulas M of* QLL*,*

$$\vdash_{\mathrm{QLL}} M \iff \models_{\mathrm{QLL}} M$$

Detailed proofs of the above results can be found in [FW97].

8.1 Canonical Constraint Models for LLP

LLP is a similarly simple fragment of QLL as logic programming (LP) is of intuitionistic predicate calculus. It is known that LP is semantically decided by a simple one-world (*i.e.* classical) canonical Herbrand model. In this section we show that for LLP, too, a simple but non-classical model can be built on the canonical constraint frame

$$\mathcal{F}: \quad \begin{tikzcd} & 1 & \\ 0 \arrow[ur, dashed] \arrow[dr, dashed] & & \\ & 2 \arrow[r, dashed] & 3 \text{ fallible} \end{tikzcd}$$

where all arrows are R_m accessibilities and world 3 is fallible. Each of the three non-trivial worlds $i = 0, 1, 2$ in the canonical model represents a different aspect of truth, and in each case this truth has a model-theoretic and an equivalent proof-theoretic side. Accordingly, we distinguish between i-derivability and i-validity and the canonical model of a LLP program has the property that a formula is i-valid iff it is i-derivable. No need to stress, the difference between the worlds $0, 1, 2$ takes account of the modal constraint operator and solvability. Specifically, the semantics is set up such that a ground atomic proposition A is forced in the model according to the following intuition:

$$\begin{array}{lll} 0 \models A & \text{iff} & A \text{ is true (in the least Herbrand model)} \\ 1 \models A & \text{iff} & \bigcirc A \text{ is true} \\ 2 \models A & \text{iff} & \bigcirc A \text{ is true and solvable} \\ 3 \models A & & \text{always (fallible world).} \end{array}$$

The semantic difference between A and $\bigcirc A$ being true implies the existence of at least two worlds, hence the nonclassical nature. For if $\bigcirc A$ is true at 0 but A false, then there must be a modally accessible world at which A is true, and this world must be different from 0. The difference between 1 and 2 is to account for the fact that not all A with $\bigcirc A$ being true have proofs with solvable constraints, *i.e.* are solvable. The purpose of the final fallible 3 is to mark its predecessor 2 as the world with solvable constraints. For if A is valid at 1 but not at 2, *i.e.* $\bigcirc A$ true at 0 but unsolvable, then $\bigcirc A$ is valid at 0 only because of the existence of 3, hence it is true in a fallible sense. In other words: by adding or deleting the fallible world 3 from the model, we can tell the difference between solvable and unsolvable constraints. Thus, the fallible world is an essential ingredient in our canonical model.

The following model construction is relative to a given language $\mathcal{L}$ and a notion of constraint $\mathcal{C} = (\Phi, \sim)$, which are are assumed to be fixed throughout the remainder of this section. As usual, the *Herbrand universe* is the set $\mathcal{H}$ of all ground terms of the language $\mathcal{L}$. We assume that Φ is generated from a set of constraint relation symbols B, and that if B has arity n the model-theoretic semantics of B is defined as an n-ary relation R_B on $\mathcal{H}$. Every constraint $c \in \Phi$ with m free variables then defines a m-ary relation R_c on $\mathcal{H}$; c is solvable if R_c is nonempty. To capture this model-theoretic semantics on the proof-theoretic side, we assume that $\sim$ is sound and complete for constraints, *i.e.* $c \sim d$ iff $R_c = R_d$.

In the following let $\Pi = \zeta_1, \ldots, \zeta_n$ be a LLP-program without constraints (but which may contain modalised and non-modalised heads) and $\Xi = p_1, \ldots, p_n$ a constraint table for Π, *i.e.* a list of closed $\lambda_c(\mathcal{C})$-terms such that p_i is of type $|\theta_i|$, where, for simplicity, we assume $\gamma(P) = 0$ for every relation symbol, *i.e.* $|A| = \mathbf{1}$ for atomic A.

The three non-trivial worlds $0, 1, 2$, of our canonical model based on frame $\mathcal{F}$, are constructed as the least Herbrand models of three different refinements Π^0, Π^1, Π^2 of Π. To begin with, Π^0 is the program Π in which all subformulas $\bigcirc A$ (*i.e.* all modalised clause heads) have been replaced by *true*. This corresponds to the most *pessimistic* constraint interpretation $\bigcirc M := \mathit{false} \supset M \equiv \mathit{true}$. This transformation switches off all modalised clauses, essentially assuming that whatever constraints are waiting behind $\bigcirc$ they will not be solvable anyway. This is the most pessimistic view one can take. Next, Π^1 is the program obtained from Π by replacing all subformulas $\bigcirc A$ by A. This is the most *optimistic* interpretation in which all constraints are assumed to be solvable. In general we would read $\bigcirc M$ as $\mathit{true} \supset M$, which is the same as M. Finally, Π^2 is the refinement $(\Xi : \Pi)^\flat$, essentially replacing each (implicit) occurrence of $\bigcirc M$ by $c \supset M$ where c is the *actual* constraint for M as given by Ξ.

Definition 8.6 *A Kripke constraint model* (W, R_i, R_m, I, F) *is a* Herbrand constraint model *for* $\Xi : \Pi$ *if it satisfies the following properties*

1. (W, R_i, R_m, F) *is the canonical frame* $\mathcal{F}$

2. $\mathsf{dom}(\alpha)$ *for all* $\alpha \in W$ *is the constant domain* $\mathcal{H}$*,* I^α *is the standard Herbrand interpretation on function, constant, and relation symbols, and* $I^\alpha(B) = R_B$ *for the constraint relations*

3. *For* $i \in W = \{0, 1, 2\}$*,* $i \models \Pi^i$

Lemma 8.7 *Every Herbrand constraint model* $\mathcal{M}$ *of* $\Xi : \Pi$ *is a model of the abstract program* Π*, i.e.* $0 \models \Pi$ *in* $\mathcal{M}$*.*

Since the program Π^i, for $i = 0, 1, 2$, is an ordinary (*i.e.* modal-free) Prolog program it has a least Herbrand model $\mathcal{M}^i$ in which the built-in constraint predicates B are assigned their predefined semantics R_B. This follows from standard results. We can now obtain the canonical Herbrand constraint model $\mathcal{M}(\Xi : \Pi) = (\mathcal{F}, I_{\Xi:\Pi})$ on the canonical frame $\mathcal{F}$ by putting $I^i_{\Xi:\Pi} = \mathcal{M}^i$ for all $i \in \{0, 1, 2\}$. One can show that this interpretation $I_{\Xi:\Pi}(i)$ satisfies the monotonicity conditions of Definition 8.1.

Lemma 8.8 $\mathcal{M}(\Xi : \Pi)$ *is a Herbrand constraint model for* $\Xi : \Pi$*.*

We wish to link the canonical worlds with canonical proof-theoretic properties of the program.

Definition 8.9 *A ground atom* A *is called a*

- 0-consequence *(of* $\Xi : \Pi$*) if there exists a derivation* $\Pi \vdash_{\text{LLP}} A$
- 1-consequence *if there exists a derivation* $\Pi \vdash_{\text{LLP}} \bigcirc A$
- 2-consequence *if it is a* 1*-consequence with proof* $\vec{w} : \Pi \vdash_{\text{LLP}} q : \bigcirc A$ *so that* $q^* := |q| \, \{\Xi/\vec{w}\} *$ *is solvable.*

Theorem 8.10 (Soundness/Completeness) *For every ground atom* A *and* $i = 0, 1, 2$ *we have:* A *is an* i*-consequence of* $\Xi : \Pi$ *iff* $i \models A$ *in* $\mathcal{M}(\Xi : \Pi)$*.*

Finally we remark that it should be possible to enrich the canonical models so that from the structure of the worlds and the forcing of atomic formulas also the solutions can be identified. For instance, we conjecture that the so-called *s-semantics* for logic programming [FLMP89] can be cast as a canonical Kripke constraint model in which worlds represent substitution constraints with free variables.

9 Points of Discussion

This work introduced Quantified Lax Logic (QLL) as a formal framework for constraint logic programming in which the extensional aspect (*what does it mean?*) and the intensional aspect (*how is it computed?*) are naturally combined. In QLL we adopt the identifications

$$\begin{array}{rcl} \text{abstract programs} & = & \text{formulas} \\ \text{constraints} & = & \text{proofs,} \end{array}$$

which accommodates the contextual nature of constraints, as opposed to programs, naturally by an inversion of direction: proofs accumulate bottom-up while formulas are refined top-down. Using this simple idea we obtain a compositional theory of constraints and logic programs in which every program construct is justified as a logic operator, both in an extensional and an intensional sense. The extensional semantics of formulas is captured by Kripke constraint models for QLL, and the extensional semantics of proofs by models of the computational lambda calculus $\lambda_c^{\Sigma\Pi}$, which correspond to the denotational or declarative semantics of CLP. The operational or procedural semantics of CLP, on the other hand, corresponds to a logic calculus and proof-search strategy for QLL. This captures the intensional aspects of executing programs and generating answer constraints. Generally, we propose the correspondences

$$\begin{array}{rcl} \text{denotational semantics} & = & \text{model theory} \\ \text{operational semantics} & = & \text{proof theory.} \end{array}$$

The proof-theoretic semantics $\lambda_c^{\Sigma\Pi}$ of QLL, presented in this report, extends Moggi's computational lambda calculus λ_c by dependent types, and the model-theoretic semantics extends the intuitionistic Kripke constraint models, previously introduced by the first two authors, by first-order universes. For the Kripke constraint models we give a soundness and completeness result by embedding QLL into classical bi-modal [S4, S4] logic. This extends the well-known Gödel embedding for intuitionistic logic into S4. Our framework captures concrete notions of constraint as different semantics of the computational lambda calculus $\lambda_c^{\Sigma\Pi}$ and different classes of Kripke constraint models. In this way, we obtain parametrisation in the domain of computation and constraint $\mathcal{C}$ and establish a new interpretation of the CLP($\mathcal{C}$) scheme.

We have defined a special class of $\lambda_c^{\Sigma\Pi}$-models, the calculi of constraint relations Λ and an even more specific family of computational lambda calculi $\lambda_c(\mathcal{C})$ corresponding to the standard CLP model. To be more precise, CLP($\mathcal{C}$) actually corresponds to the Lax Logic Programming fragment LLP of QLL. For LLP we presented a class of 4-world constraint Kripke models, and the logic calculi $\vdash_r$, $\vdash_l$ as two different formalisations of constrained SLD resolution. In specialising to LLP we obtain a natural operational and denotational semantics for constraint logic programming with an accompanying notion of constraint abstraction and constraint refinement. Abstraction amounts to splitting a concrete CLP program into a *proof : formula* pair. This separates concerns in a logically very precise sense and reinterprets the characteristic feature of CLP, which is to consider constraints as built-in predicates, in a new way. Refinement, then, is the recombination of constraints and abstract program. We have given concrete abstraction and refinement mappings for LLP programs adequate for CLP.

9.1 Related Work

The work presented here is not the first proof-theoretic approach to the semantics of LP or CLP. Other approaches in the proof-theoretic tradition were made by Hagiya and Sakurai [HS84], Hallnäs and Schroeder-Heister [HSH90], Andrews [And92], Darlington and Guo [DG94]. These semantics essentially are *provability* semantics in which one replaces the model-theoretic definition of program behaviour as the class of queries true in all Herbrand models by a formal-deductive definition, *viz.* as the class of queries provable in some calculus. Although these semantics refer to the notion of proof they are still extensional since what is considered relevant is *that* a query is derivable and not *how* it is derivable. In other words, it does not matter which calculus one uses as long as it derives the same theorems. This is quite different with the intensional framework presented here in which a particular choice of a (sub-structural) calculus for QLL determines a particular method of generating a particular class of answer constraints.

The extensional point of view is applied, in particular, in the work of Andrews [And92] which presents a proof-theoretic analysis of different operational semantics of LP. He shows that the classes of successful and failing queries for several versions of parallel/sequential and/or style operational semantics can be characterised by provability in different logic calculi. These calculi generate classical theories with logic modalities $S(A)$ and $F(A)$ to specify the success and failure of queries A, respectively. They constitute axiomatic semantics for several operational models, very much like the Floyd-Hoare axiomatics for imperative programs. This is an *external* semantic characterisation which keeps a clear distinction between Prolog queries as the object level and the success/failure formulas specifying the semantics of these objects. For this external approach it is accidental, so to speak, that Prolog programs themselves can be seen as logic formulas. In contrast, here we are interested in the proof theory of logic programs themselves, *i.e.* their *internal* logic and *internal* proof theory. This, so we believe, takes better account of the logical nature of logic programming, and is more appropriate to capture operational aspects of constraint logic programming, in a proof-theoretic way.

The problem of combining theorem-proving with constraint generation has been addressed also in the work of Hallnäs and Schroeder-Heister [HSH90] in which it was proposed to replace the standard sequent calculus by a new higher-level calculus LC(P) to formalise the generation of non-ground answer substitutions. However, this does not solve the problem, since LC(P) is not a logic calculus with an independent model-theoretic semantics, but rather a formal presentation of an operational semantics for a programming language that happens to be a logic. In this sense, this work is similar in flavour to the approach of Andrews.

The work of Darlington and Guo [DG94] is related to ours in that they, too, link up the operational mode of constrained SLD resolution for CLP with provability in a constructive sequent calculus. Their calculus of simple uniform proofs is comparable to our $\vdash_r$ system. However, again, the correspondence verified by Darlington and Guo is of the extensional kind, which does not address the operational issue of actually constructing constraints, which ours does. From this point of view constrained SLD resolution is not fully formalised by the system of Darlington and Guo, strictly speaking.

The operational issue of constraint generation, though not by proof-theoretic means, is addressed by Darlington, Guo, and Wu in [DGW91]. This work, just like ours, gives a separation of the constraint generation from the deduction procedure, by implicit abstraction. They use context-free grammars as an abstract representation of CLP programs that are stripped of constraints. The deduction mechanism of CLP, thus, is translated into executing the production rules of a context-free grammar, and from the generated words, in [DGW91] called *solving plans*, constraints are extracted. The context-free grammar obtained from a program corresponds to our notion of an abstracted program, where we abstract completely from all constraints and first-order terms. Hence, it corresponds to complete propositional abstraction. In our framework abstraction need not be as drastic as that, we can leave some of the program structure untouched. If only partial abstraction is to be used, *e.g.* if only constraint formulas are to be abstracted and all first-order structure is kept, then simple grammar production like in [DGW91] does not suffice to model the execution of the abstract program. We also need first-order unification. Thus, our notion of constraint and constraint generation is more flexible. It is defined by a more flexible notion of abstraction, *i.e.* a more fined-grained notion of observation. It allows us to adjust more carefully which parts of the program we wish to observe and which not. That is, how much work is to be taken away from the inference engine and delegated to the constraint engine. Blunt propositional abstraction is but one extreme case.
What the solving plans of [DGW91] are the $\lambda_c^{\Sigma\Pi}$-proof terms in our framework. Being generated by the execution of an abstract program they determine constraints by semantic translation. Comparing the two formalisms one observes that both maintain a different amount of structural information about constraints. Solving plans are linear sequences of terminal symbols; each terminal represents (the invocation of) a primitive constraint and the whole sequence is a conjunction of (invocations of) such primitive constraints. More structure, in contrast, is present in our proof terms. The $\lambda_c^{\Sigma\Pi}$ terms have a tree-like structure of nested $\mathsf{let\text{-}in}$ variable bindings and function applications which retains the hierarchy and potential parallelism inherent in the execution of conjunctive subgoals. In the solving plans of [DGW91] this hierarchical structure is flattened and the parallelism is sequentialised. We imagine that this extra structure of $\lambda_c^{\Sigma\Pi}$, in certain cases, can be exploited to compute constraints more directly. This is not explored here, but an example of this idea is given [Men96] which applies Propositional Lax Logic to extract timing information for combinational circuits. There it is shown that propagation delays can be obtained immediately by reading proofs as expressions in the max-plus algebra, rather than as conjunctions of timing inequations, which must be solved first to get hold of the propagation delay. Another advantage

of the $\lambda_c^{\Sigma\Pi}$ calculus is its built-in invocation handling. The lambda calculus with its variable abstraction and function application mechanism serves to organise the instantiating of constraints and renaming of variables in a systematic and consistent way. In [DGW91] this problem is not addressed. Finally, we note that the translation of $\lambda_c^{\Sigma\Pi}$ terms into concrete constraints (via $\lambda_c(\mathcal{C})$) in general introduces existential quantification which keeps some of the variables local. When full propositional abstraction is used, like in [DGW91], then all variables are existentially quantified. This localisation is important for efficient constraint solving.

9.2 Critical Remarks and Future Work

If we were to sum up this work in one sentence it would essentially propose the following programme: Use a constructive logic as a generic constraint logic programming system, capturing, in a single language, both denotational and operational semantics. Although this report does not get very far with this programme it does attempt to pin down what might be considered the beginnings of such a general theory. Nevertheless, the authors are well aware that the technical apparatus introduced here may seem excessively involved merely to capture the relatively simple deductive mechanism of ordinary CLP semantics. An obvious disadvantage, from an implementation point of view, is the fact that in order to realise the decoupling of abstract programs and constraints we need to generate and carry around (partial) proof objects. This is computationally expensive and will be worthwhile only in applications where constraints actually need to be delayed in the course of a program execution, say to free the main thread of computation from the need to check constraint solvability immediately, and to operate ahead in exploring the search space at a more abstract level. This technique is well established in theorem proving but less so in constraint logic programming.

The material presented in this report does not attempt to contribute towards the algorithmic side of CLP. Its contribution lies in presenting a new mathematical framework in which to study and compare CLP semantics from a logical point of view. We believe that this approach will prove useful as a rigorous basis for more ambitious notions of constraint and CLP paradigms that narrow the gap to full-fledged intuitionistic theorem-proving. It should be possible also to treat CLP schemes with negation, inductive reasoning, or higher-order clauses in much the same way, based on (extensions of) QLL and $\lambda_c^{\Sigma\Pi}$.

More work is needed to investigate the correspondence between proof search and proof analysis in QLL and different operational semantics for CLP. The reader will have noticed that in the work presented here we evaded discussing the issue of checking satisfiability of constraints, which is an important part of standard CLP systems. This is justified since, in our opinion, it is more of a challenge to find satisfactory ways for separating user program and constraints without jeopardising correctness, than to mix them up in a single powerful calculus. This latter problem is easy and has been solved already, the former however has not been addressed sufficiently. Granted that, the question still remains: How can we incorporate the satisfiability check? Well, in our framework the satisfiability check is a form of proof analysis. Technically, it corresponds to proof normalisation, which can be performed, in principle, at any time in the course of a logic derivation. The idea is that $\rightarrow_{\beta c \gamma}$ reduction, which now also includes reduction of constraints to solved form, if it succeeds, establishes the solvability of the constraint represented by the proof term. This is an intriguing and mathematically convincing way to include constraint solvability into our framework, which deserves to be explored in future work. In this context the "termination" predicate of evaluation logic [Pit91, Mog95] suggests itself as a formal way to introduce solvability alias normalisability into the language of propositions, and to axiomatise constraint solvability within QLL.

Future work also will explore several avenues of extending the CLP($\mathcal{C}$) scheme suggested by our framework. We propose to investigate more powerful abstraction mechanisms to generate models of QLL and apply recent work into combining and cascading monads to obtain a logic theory of combined and iterated levels of abstractions and constraints. A potential application is the abstract analysis and debugging of logic programs. We envisage that user interactions can also be captured as a notion of constraint along the lines sketched in this work, just as input/output is formalised semantically by monads in functional programming languages such as Haskell [Tho96].

References

[And92] J. H. Andrews. *Logic Programming: Operational Semantics and PRoof Theory*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1992.

[AV96] M. Argenius and A. Voronkov. Semantics of constraint logic programs with bounded quantifiers. In R. Dyckhoff, H. Herre, and P. Schröder-Heister, editors, *Extensions of Logic Programming*, volume 1050 of *Lecture Notes in Artificial Intelligence*, pages 1–18. Springer-Verlag, 1996.

[BBdP95] N. Benton, G. Bierman, and V. de Paiva. Computational types from a logical perspective I. Technical Report 365, University of Cambridge Computer Laboratory, May 1995.

[Cla78] K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, New York, 1978. Plenum Press.

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

[Col87] A. Colmerauer. Opening the Prolog III universe. *BYTE magazine*, pages 177–182, August 1987.

[Col90] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[DG94] J. Darlington and Y. Guo. Constraint logic programming in the sequent calculus. In F. Pfenning, editor, *Logic Programming and Automated Reasoning*, pages 200–214. Springer LNAI 822, 1994.

[DGW91] J. Darlington, Y. Guo, and Q. Wu. A general computational scheme for constraint logic programming. In *Proc. 3rd U.K. Annual Conference on Logic Programming*, pages 56–77. Springer, 1991.

[DvHSA88] M. Dincbas, P. van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language CHIP. In *Proc. Second International Conference on Fifth Generation Computer Systems*, pages 249–264, 1988.

[FHK+92] T. Frühwirth, A. Herold, V. Küchenoff, T. Le Provost, E. Monfray, and M. Wallace. Constraint logic programming – an informal introduction. In G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, editors, *Proc. Logic Programming in Action, Second International Logic Programming Summer School (LPSS)*, volume 636 of *Lecture Notes in Computer Science*, pages 3–35. Springer-Verlag, 1992.

[FLMP89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69:289–318, 1989.

[FM95] M. Fairtlough and M. V. Mendler. An intuitionistic modal logic with applications to the formal verification of hardware. In *Proceedings of the 1994 Annual Conference of the European Association for Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 354–368. Springer-Verlag, 1995.

[FM97] M. Fairtlough and M. V. Mendler. Propositional Lax Logic. *Information and Computation*, 137(1):1–33, August 1997.

[FW97] M. Fairtlough and M. Walton. Quantified Lax Logic. Technical Report CS-97-11, Department of Computer Science, University of Sheffield, 1997.

[GDL95] M. Gabrielli, G. M. Dore, and G. Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

[Göd69] K. Gödel. An interpretation of the intuitionistic sentential logic. In J. Hintikka, editor, *The Philosophy of Mathematics*. Oxford University Press, 1969.

[GW92] F. Giunchilglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.

[HS84] M. Hagiya and T. Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2(1):59–77, 1984.

[HSH90] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. I. clauses as rules. *Journal of Logic and Computation*, 1(2):261–283, 1990.

[JL87] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119. ACM, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19(20):503–581, 1994.

[JMMS96] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programs. Technical Report 96/39, University of Melbourne, November 1996.

[JMSY92] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[Kri92] F. Kriwaczek. An introduction to constraint logic programming. In V. Mařík, O. Štěpánková, and R. Trappl, editors, *Proc. Advanced Topics in Artificial Intelligence, International Summer School*, volume 617 of *Lecture Notes in Artificial Intelligence*, pages 82–94. Springer-Verlag, 1992.

[Las87] C. Lassez. Constraint logic programming. *BYTE Magazine*, 12, August 1987.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987. Extended.

[MB91] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design.* Kluwer, 1991.

[Men90] M. Mendler. Constrained proofs: A logic for dealing with behavioural constraints in formal hardware verification. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits (DCC '90)*, pages 1–28. Springer, 1990.

[Men93] M. Mendler. *A Modal Logic for Handling Behavioural Constraints in Formal Hardware Verification.* PhD thesis, Edinburgh University, Department of Computer Science, ECS-LFCS-93-255, 1993.

[Men96] M. Mendler. Timing refinement of intuitionistic proofs and its application to the timing analysis of combinational circuits. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 261–277. Springer-Verlag, 1996.

[MF96] M. Mendler and M. Fairtlough. Ternary simulation: A refinement of binary functions or an abstraction of real-time behaviour? In M. Sheeran and S. Singh, editors, *Proceedings of the 3rd Workshop on Designing Correct Circuits (DCC'96), Båstad, Sweden.* Springer Electronic Workshops in Computing, September 1996.

[Mog88] E. Moggi. *The partial lambda calculus.* PhD thesis, Edinburgh University, Department of Computer Science, August 1988.

[Mog90] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, 1990.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mog95] E. Moggi. A semantics for evaluation logic. *Fundamenta Informaticae*, 22:117–152, 1995.

[MSea93] M. Meier, J. Schimpf, and et al. ECL^{i}PS^e: ECRC common logic programming system, user manual. Technical Report TTI/3/93, ECRC, 1993.

[Pit91] A. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher-Order Workshop, Banff 1990*. Springer, 1991.

[Pla81] D. A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[Pop94] S. Popkorn. *First Steps in Modal Logic*. Cambridge University Press, 1994.

[Pou95] D. Pountain. Constraint logic programming. *BYTE Magazine*, February 1995.

[SA89] K. Sakai and A. Aiba. CAL: A theoretical background of constraint logic programming and its application. *J. Symbolic Computation*, 8:589–603, 1989.

[Sch89] B.-A. Scharfstein. *The Dilemma of Context*. New York University Press, 1989.

[Tho96] S. Thompson. *Haskell - The Craft of Functional Programming*. Addison-Wesley, 1996.

[TU93] E. Tyugu and T. Uustalu. Higher-order functional constraint networks. Technical report, The Royal Institute of Technology, Department of Teleinformatics, November 1993.

[van89] D. van Dalen. *Logic and Structure*. Springer-Verlag, 1989.

[Wad90] P. Wadler. Comprehending monads. In *Conference on Lisp and Functional Programming*, pages 61–78. ACM Press, June 1990.

[Wad92] P. Wadler. Monads for functional programming. In *Lecture Notes for the Marktoberdorf Summer School on Program Design Calculi*. Springer-Verlag, August 1992.