HYPROLOG: A New Logic Programming Language with Assumptions and Abduction

Henning Christiansen[1] and Veronica Dahl[2]

[1] Roskilde University, Computer Science Dept., Roskilde, Denmark
[2] Dept. of Computer Science, Simon Fraser University Burnaby, B.C., Canada
henning@ruc.dk, veronica@cs.sfu.ca

Abstract. We present HYPROLOG, a novel integration of Prolog with assumptions and abduction which is implemented in and partly borrows syntax from Constraint Handling Rules (CHR) for integrity constraints. Assumptions are a mechanism inspired by linear logic and taken over from Assumption Grammars. The language shows a novel flexibility in the interaction between the different paradigms, including all additional built-in predicates and constraints solvers that may be available. Assumptions and abduction are especially useful for language processing, and we can show how HYPROLOG works seamlessly together with the grammar notation provided by the underlying Prolog system. An operational semantics is given which complies with standard declarative semantics for the "pure" sublanguages, while for the full HYPROLOG language, it must be taken as definition. The implementation is straightforward and seems to provide for abduction, the most efficient of known implementations; the price, however, is a limited use of negations. The main difference wrt. previous implementations of abduction is that we avoid any level of metainterpretation by having Prolog execute the deductive steps directly and by treating abducibles (and assumptions as well) as CHR constraints.

1 Introduction

Assumption-based reasoning in general, or hypothetical reasoning is defined in [22] as a logic system in which a set of facts and a set of possible hypotheses are given. Its instances can be assumed if they are consistent with the facts. Both abduction (the unsound but useful assumption of B given A and given that B implies A) and linear and intuitionistic logic inspired assumptions (special facts that are made available as global resources within a specific scope [25]) fall into that general category. Their formalization within, respectively, Abductive Logic Programming [19] and Assumptive Logic Programming [14] refines this general notion by for instance requiring in the first case consistency with a special type of facts: integrity constraints. Both allow us to move beyond the limits of classical logic to explore "possible cause" and "what-if" scenarios. They have proved useful for diagnosis, recognition, sophisticated human language processing problems, and many other applications. However in practice, abduction in particular

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 159–173, 2005.

has not been used to its full potential owing to implementation indirections. Assumptions can be more efficiently implemented through continuation based processors such as BinProlog, but there is no Prolog in existence which efficiently provides both capabilities at the same time.

The present paper generalizes and improves an earlier proposal presented in the workshop paper [12]. In this article we present a new programming language, HYPROLOG, which augments Prolog with the following hypothetical reasoning capabilities:

- linear, intuitionistic and timeless assumption, in the sense of [14] to which we add the new feature of integrity constraints,
- abduction in the sense of abductive logic programming [19],
- integrity constraints that may refer to both abducibles and assumptions.

These results are significant in that they enhance Prolog's appeal as a programming language by transporting it beyond the rigid limits of classical logic and thus making it more appropriate for human-like reasoning in general and for AI in particular. In addition, they are *portable*, in the sense that programs in any Prolog that includes CHR can simply be augmented with our code, which provides the mentioned extensions, and *efficient*: the assumptive part runs only three times slower than in Prolog versions where assumptions are hardwired, and our abduction runs actually faster than previous implementations for programs that involve many resolution steps. Finally, our implementation principles can be seen as demonstration of how hypothetical reasoning can materialize in CHR even without our system.

We first overview the necessary background on abduction, assumptions, and CHR; then we present the new language's syntax and exemplify its use within both programs and grammars. We discuss implementation principles, semantic considerations, related work and benchmarks, and finally, we provide concluding remarks. System and sample programs are available at
http://www.ruc.dk/~henning/hyprolog.

2 Background

2.1 Abduction

An abductive logic program [19] is usually specified as a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of *abducible* predicates that do not occur in the head of any clause of $\mathcal{P}$, and $\mathcal{IC}$ a set of integrity constraints assumed to be consistent. Assume additionally that $\mathcal{P}$ and $\mathcal{IC}$ can refer to a set of *built-in* predicates that have a fixed meaning identified as a theory $\mathcal{B}$; a predicate in $\mathcal{P}$ that is neither abducible nor built-in is called *defined*. We assume for simplicity in the following that $\mathcal{IC}$ refers to abducible and built-in predicates only.

Given an abductive logic program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$, we define for pairs of sets of abducibles and built-in atoms $\langle A, B \rangle$, a *consistent ground instance* to be a common ground instance $\langle A', B' \rangle$ of $\langle A, B \rangle$ so that

– $\mathcal{B} \models B'$ (the instance of built-ins is satisfied)
– $\mathcal{B} \cup A' \models \mathcal{IC}$ (the instance of abducibles respects the integrity constraints)

For simplicity and without loss of generality, we consider only ground queries; an *abductive answer* to a query Q is a pair of finite sets of abducible and of built-in atoms $\langle A, B \rangle$ such that

– $\langle A, B \rangle$ has at least one consistent ground instance $\langle A', B' \rangle$,
– for any such $\langle A', B' \rangle$, we have $\mathcal{P} \cup A' \models Q$.

Minimality and Compaction. It is often required that an abductive answer be minimal measured in the number of abduced literals (or, alternatively, in a subset relation or subsumption ordering). Most published abduction algorithms try to unify a new abducible with one already produced (as to produce answers of a minimum number of literals), and tries out different alternatives under backtracking. This does not guarantee minimality in cases when, say, a proof needs abducibles a and b but another may need only a. Minimal answers can be selected by post-processing all answers found in this way. However, we argue that this principle which we call *compaction* is not always obvious or desirable, and we suggest it be optionally specified for selected abducible predicates. (If, for example, someone's car was stolen in Paris and his wallet in New York, it seems over-constrained to assume by default that the thieves are the same one.)

2.2 Assumptive Logic Programming

Assumptive logic programs [14] are logic programs augmented with a) linear, intuitionistic and timeless implications scoped over the current continuation, and b) implicit multiple accumulators, useful in particular to make the input and output strings invisible when a program describes a grammar (in which case we talk of Assumption Grammars [15]). More precisely, we use the kind of linear implications called *affine* implications, in which assumptions can be consumed at most once, rather than exactly once as in linear logic. Although intuitively easy to grasp and to use, the formal semantics of assumptions is relatively complicated, basically proof theoretic and based on linear logic [14,15,25]. Here we use a more recent and homogeneous syntax for assumptions introduced in [10]; we do not consider accumulators, and we note that Assumption Grammars can be obtained by applying the operators below within a DCG.

`+h(a)`	Assert linear assumption for subsequent proof steps. Linear means "can be used once".
`*h(a)`	Assert intuitionistic assumption for subsequent proof steps. Intuitionistic means "can be used any number of times".
`-h(X)`	Expectation: consume/apply existing int. assumption.
`=+h(a)`, `=*h(X)`, `=-h(X)`	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

A sequential expectation cannot be met by timeless assumption and vice versa, even when they carry same name. In [15], a query cannot succeed with a state which contains an unsatisfied expectation; for simplicity (and to comply with our implementation), this is not enforced in HYPROLOG but can be tested explicitly using a primitive called `expections_satisfied`. Assumption grammars have been used for natural language problems such as free word order, anaphora, coordination, and for knowledge based systems and internet applications. In the earlier work on Assumptions, only a semiformal semantics was given, and the semantics we show below are intended to make its principles precise.

2.3 Constraint Handling Rules, CHR

CHR [17] is a declarative, rule-based language for writing constraint solvers and is now included as an extension of several versions of Prolog. Operationally and implementation-wise, CHR extends Prolog with a constraint store, and the rules of a CHR program serve as rewriting rules over constraint stores. CHR is declarative in the sense that its rules can be understood as logical formulas. Constraint predicates must be declared as such and can then be called from a Prolog program; see [17] for details. The following example declares a constraint predicate `a` and defines a so-called propagation rule.

```
constraints a/1.
a(1), a(2) ==> fail.
```

This rule identifies a state as illegal if it contains the two indicated constraints. As first noticed by [3], there is a clear analogy between abducibles plus integrity constraints and CHR's constraints plus rules.

3 HYPROLOG, Syntax and Informal Semantics

3.1 Basic HYPROLOG

A HYPROLOG program is written as a Prolog program with additional declarations of assumptive and abductive predicates, the latter possibly with compaction. Notation for applying assumptions is shown in the previous section. Integrity constraints are written as any sort of CHR rules with abducibles and assumptions in the head. The following exemplifies such declarations.

```
abducibles a/1, b/2.
compaction a/1.
assumptions c/1.
timeless_assumption d/2.
```

The first declaration introduces abducible predicates `a/1`, `b/2` as well as `a_/1`, `b_/2` that represent their negation; compaction is defined for `a/1` (as described above). The declaration of `c/1` makes available assumptions and expectations of forms `'+c'/1`, `'-c'/1`, `'*c'/1` (the system reads, say, `+c(5)` as `'+c'(5)`).

3.2 HYPROLOG's Grammatical Counterpart

DCGs [21], included in most Prolog systems and compiled into Prolog when a source file is loaded, are also available in HYPROLOG, adequately augmented with abduction and assumptions as well. The following example has been adapted from [10,15] and shows two applications of assumptions: for resolving pronoun references and for a simple coordination problem. In a sentence *"Peter likes her"* the pronoun is expected to stand for a female character who has been mentioned earlier in the discourse. The following rule defines how the mention of a proper name produces an (intuitionistic) assumption that makes the individual available for future reference, as many times as needed.

```
assumptions acting/1.

np(X,Gender) --> name(X,Gender), {*acting(X,Gender)}
```

Let's suppose we have the following rules for sentences and sequences of sentences.

```
sentence(s(A,V,B)) -->  np(A,_), verb(V), np(B,_).
sentences((S1,S2)) --> sentence(S1),sentences(S2).
sentences(nil) --> [].
```

The following rules define how a pronoun can appear in a sentence with its meaning given by the consumption of an assumption made.

```
np(X,Gender) --> {-acting(X,Gender)}, pronoun(Gender).
pronoun(fem) --> [her].
```

The following query and answers show the grammar's behaviour.

```
?- phrase(sentences(S), [peter,likes,martha, mary,hates,her]).
S = (s(peter,like,martha),s(mary,hate,mary),nil) ? ;
S = (s(peter,like,martha),s(mary,hate,martha),nil) ? ;
no
```

The second answer expresses the interpretation we would expect, and the first one is an undesired consequence of the specification so far; we show below how it can be suppressed.

The discourse *"Peter likes and Mary hates Martha"* contains two coordinating sentences in the sense that the first incomplete one takes its object from the second one. This can be described by having an incomplete sentence put forward a timeless expectation that may be satisfied by a later assumption produced by a complete sentence; the following two grammar rules are sufficient.

```
sentence(s(A,V,B)) --> np(A,_), verb(V), np(B,_), {=*obj(B)}.
sentence(s(A,V,B)) --> np(A,_), verb(V), [and],   {=-obj(B)}.
```

3.3 Mixing Abduction and Assumptions

Abduction and assumptions can be mixed freely, which is handy for a better solution to the pronoun resolution problem above. We modify the grammar rule for sentences such that semantic interpretation is made abductively, i.e., the sentence can be told honestly provided the semantic context contains the necessary facts.

```
abducibles s/3.
sentence --> np(A,_), verb(V), np(B,_), {s(A,V,B)}.
s(X,hate,X) ==> fail.
```

With this modification, the analysis of *"Peter likes Martha, Mary hates her"* gives only one solution. This grammar is interesting as it shows how different layers of analysis can assist each other: semantic knowledge about the hating relation is applied for guiding pronoun resolution. This example illustrates a general approach to discourse analysis called Meaning-in-Context, described in [13].

3.4 Negation

Compared with other abductive systems, the use of negation is quite limited and restricted to a simple form of explicit negation [7]. When an abducible, say `a/1`, is declared, an additional predicate `a_/1` representing ¬`a` is introduced together with an integrity constraint (hidden from the user) `a(X), a_(X) ==> fail.`

Although useful for many applications, this implementation covers only one part of negation: "you cannot have P and $\neg P$ at the same time"; the condition that "either you have P or $\neg P$" cannot be expressed in a straightforward way.

If a program clause includes an application of negation-as-failure that refers to abducibles directly or indirectly, we inherit the dubious semantics of Prolog. So if `a/1` has been declared as abducible, with definition `p(X):-a(X)`, a call `\+p(Z)` (where `Z` is a currently uninstantiated variable) may succeed if the abduction of `a(Z)` triggers a failure producing integrity constraint.

4 Semantic Considerations

Like Prolog, CHR has a declarative semantics plus a procedural one, and for a substantial subset of the language, the two are in agreement. Each rule of a CHR program can be understood as a logical formula:

	Propagation rule	Simplification rule
CHR rule:	H==>$G\|B$	H<=>$G\|B$
Logical meaning:	$\forall\bar{x}\big((\exists\bar{y}G) \rightarrow (H \rightarrow \exists\bar{z}B)\big)$	$\forall\bar{x}\big((\exists\bar{y}G) \rightarrow (H \leftrightarrow \exists\bar{z}B)\big)$

where $\bar{x}$ refers to the variables in H, $\bar{y}$ to those in G not overlapping with $\bar{x}$, and $\bar{z}$ to those in B not overlapping with $\bar{x}$; for simplicity it is assumed that $\bar{y}$ and $\bar{z}$ do not overlap. Ignoring the problems with Prolog's negation as failure, we can say that the meaning (e.g., a model-based semantics) of a program that

combines Prolog and CHR is given by formulas as above plus a reading of the Prolog part as a completed definition.

However, as has been debated recently [16], the statements that can be made by using this semantics for CHR are often too weak to express the (implemented) meaning of even simple and intuitively clear programs. Even the classical, procedural semantics with nondeterminism in selection steps [2] is not sufficient; as noted by [16], even example programs in the reference manual of CHR depend on the implemented semantics, and this motivated [16] to describe a so-called refined procedural semantics.

The part of HYPROLOG without assumptions is an instance of abductive logic programs and conforms with the standard semantics given in section 2.1; this is independent of whether the logically redundant compaction principle is applied. However, as we use CHR for integrity constraints, the discussion above goes for this subset of HYPROLOG as well.

Assumptions, on the other hand, inherit the procedural flavour of linear logic, and a correct semantics for HYPROLOG without abduction, and even without integrity constraints, needs to reflect a left-to right execution of the clause bodies. In other words, the comma cannot be understood as conjunction but as a sequential operator that pushes a perhaps modified state forward.

Interestingly, [5] has proposed recently a semantics of CHR formulated in terms of linear logic, and a very interesting next step could be to generalize this for HYPROLOG. This may perhaps provide a more straightforward characterization of the assumption part of HYPROLOG, as assumptions can be mapped to their natural counterpart in linear logic. This possibility has not been investigated yet.

4.1 A Continuation Semantics for HYPROLOG

Here we specify the CHR engine and its constraint stores as an abstract data type (whose detailed specification can be found in [16]). For simplicity we assume only one built-in which can be used in clause bodies, "=" with the meaning of equality (unification). What may be allowed in bodies of integrity constraints is abstracted away. By a *constraint*, we mean an abducible atom, an atom of a built-in predicate, an assumption, or a timeless expectation; let Con refer to the set of all such. We assume a given HYPROLOG program of clauses $C\ell$ and integrity constraints IC.

Let $Store$ be a sort for all possible constraint stores whose internal structure is not specified and which is equipped with the following operations; Sub refers to the domain of substitutions; mgu is used to denote a most general unifier of two atoms. When relevant, a substitution σ can also be understood as a set of equations $\{x = t \mid x\sigma \text{ evaluates to } t\}$. The following operations are assumed:

- $\in$: $Con \times Store \rightarrow \{true, false\}$.
- $\setminus : Con \times Store \rightarrow Store$ representing the removal of a constraint from the store.
- $Accommodate : Con^* \times Store \rightarrow Store \times Subst$ corresponding to the CHR engine's behaviour given IC when one or more constraints are called from

a clause; whatever recursion takes place inside *Accommodate* is not specified; the output substitution represents possible side-effects that affect the remaining query. The function is partial, undefined meaning failure or loop.
- $\emptyset : Store$ which is the initial store.

Notice that the definitions allow *Accommodate* to take also a substitution as its first argument. This reflects the property that the unification of variables may trigger CHR rules to apply. This abstract data type is assumed to be *sound* in the sense that, whenever $Accommodate(A, S) = \langle S', \sigma \rangle$, we have that $IC \models \forall \bar{x}((A \wedge S) \leftrightarrow \exists \bar{z}(S' \wedge \sigma))$ where here a store is identified with the set of all constraints in it (given by $\in$); $\bar{x}$ are the variables in $A \wedge S$ and $\bar{z}$ any remaining variables in $S' \wedge \sigma$. For simplicity, let us ignore the risk of loops and also claim it *complete* meaning that $Accommodate(A, S)$ is defined whenever $IC \models \exists \bar{x}(A \wedge S)$. In case no integrity constraints are involved, the constraint store serves as a passive container for abducibles and assumptions.

A *query* is a sequence of atoms; ϵ is the empty query; concatenation and construction of sequences are indicated by a dot, and for readability the comma of the clause syntax is taken as dot. A *state* is a pair of a query and store. A *final* state is of the form $\langle \epsilon, S \rangle$. A *derivation* for Q is produced from a finite number of derivation steps (below), starting from $\langle Q, \emptyset \rangle$, and it is *successful* if it ends in a final state. The derivation relation $\rightsquigarrow$ is defined by the following rules.

1. $\langle A \cdot Q, S \rangle \rightsquigarrow \langle (B \cdot Q)\sigma\sigma', S' \rangle$ if there is a program clause which has a variant H `:-` B with fresh variables, $\sigma = mgu(H, A)$ and $Accommodate(\sigma, S) = \langle \sigma', S' \rangle$.
2. $\langle s = t \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever $mgu(s, t) = \sigma$ and $Accommodate(\sigma, S) = \langle \sigma', S' \rangle$.
3. $\langle Ab \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever Ab is a compacting abducible and there is some $A \in S$ with $mgu(A, Ab) = \sigma$ and $Accommodate(\sigma, S) = \langle \sigma', S' \rangle$
4. $\langle Ab \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma, S' \rangle$ whenever Ab is an abducible and $Accommodate(Ab, S) = \langle \sigma, S' \rangle$.
5. $\langle$`-`$A \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever, either
   - there is a `+`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{$`+`$A'\}) = \langle S', \sigma' \rangle$, or
   - there is a `*`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S) = \langle S', \sigma' \rangle$.
6. $\langle As \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma, S' \rangle$ where $Accommodate(As, S) = \langle S', \sigma \rangle$, As of form `+`A or `*`A.
7. $\langle$`=-`$A \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ whenever, either
   - there is an `=+`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{$`=+`$A'\}) = \langle S', \sigma' \rangle$,
   - there is an `=*`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S) = \langle S', \sigma' \rangle$, or
   - $Accommodate($`=-`$A, S) = \langle S', \sigma' \rangle$ and $\sigma = \emptyset$.
8. $\langle$`=+`$A \cdot Q, S \rangle \rightsquigarrow \langle Q\sigma\sigma', S' \rangle$ where
   - there is an `=-`$A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{$`=-`$A'\}) = \langle S', \sigma' \rangle$, or
   - $Accommodate($`=+`$A, S) = \langle S', \sigma' \rangle$ and $\sigma = \emptyset$.

9. $\langle \texttt{=*}A \cdot Q, S\rangle \rightsquigarrow \langle Q'\sigma\sigma', S'\rangle$ where
   - there is an $\texttt{=-}A' \in S$ with $mgu(A, A') = \sigma$ and $Accommodate(\sigma, S \setminus \{\texttt{=-}A'\}) = \langle S', \sigma'\rangle$, and Q' is one of $\texttt{=*}A \cdot Q$ or Q, or
   - $Accommodate(\texttt{=*}A, S) = \langle S', \sigma'\rangle$, $\sigma = \emptyset$, and $Q' = Q$.

We notice that step 1 defines an operational semantics for the pure Prolog subset of HYPROLOG, 1–2 for pure Prolog with built-in equality, and 1–4 (or, alternatively, 1–2 plus 4) for abductive logic programs (with CHRs as ICs and no interesting negation); these operational semantics are straightforward to prove sound and complete with respect to the respective declarative semantics. Step 1 plus 5–9 provides a formal semantics for Assumptive Logic Programs which has been lacking in earlier references.

In the lack of a truly declarative semantics (first-order or otherwise) for assumptions, we need to take 1–9 as *the* semantic definition for the full HYPROLOG language.

Finally, we notice that the actually implemented HYPROLOG system inherits the detailed procedural semantics of Prolog (trying rules in textual order) and of CHR (cf. the refined semantics [16]), both of which were abstracted away above. Being hosted in a realistic version of Prolog which includes a realistic version of CHR, all other low and high level features of these languages are available, including Prolog's negation as failure (with the usual caveats) and large collections of constraint solvers and built-in predicates.

5 Implementation

Our implementation uses SICStus Prolog [23] and its CHR library; we refer to the proper sections of the referenced manual for a detailed description of the facilities that we use. The principles shown can also be used for implementing various kinds of hypothetical reasoning in Prolog through CHR.

5.1 Implementing Abduction

The implementation in Prolog with CHR is simple: abducibles are viewed as constraints in the sense of CHR: the logic program is executed by the Prolog system; whenever an abducible is called it is added automatically by CHR to the constraint store and CHR will activate integrity constraints whenever relevant. The complete hand-coded implementation of an abducible predicate `a/1` is provided by the following three lines.

```
:- use_module(library(chr)).
handler abduction.
constraints a/1.
```

Compaction for `a/1` is implemented by a single CHR rule; the following provides a correct implementation.

```
a(X), b(Y) ==> true | (X=Y ; dif(X,Y)).
```

(The implementation of HYPROLOG applies a slightly optimized version using low-level facilities of CHR.) When a HYPROLOG program is read from file, declarations as shown in section 3.1 are translated into CHR as shown here.

The correctness is inherited from the correctness properties of the underlying Prolog plus CHR systems. For any program without occurs-check problems, the implementation produces correct abductive answers as defined above; if the program (including integrity constraints) does not loop, we also have that the total set of answers produced is complete.

Notice that the approach can interact with an arbitrary constraint solver by considering its constraints as built-ins (applied in bodies of clauses and integrity constraints). Possible soundness and completeness of such a combination will mirror the properties of the applied constraint solver.

5.2 Implementing Assumptions

Assumptions and expectation operators are implemented in CHR in a way similar to abduction, but need extra care for scoping and matching of expectations with assumptions. Each operator for each declared assumption (say `'-c'/1` is implemented by one single-headed CHR rule that employs the constraint store as container in a straightforward procedural way, optimized using the low-level primitive `findall_constraints` and `remove_constraints`; see the HYPROLOG website for details.

6 Examples and Benchmarks

Suppose we need to schedule the printing jobs of three printers. At any time, the status of each printer is represented by an assumption `+printer(` *name*`,` *ready-time*`)`.

Further, assume that all printers are covered by the same undersized electrical fuse that will melt down in case all three printers are running at the same time. Such situations is prevented by the following integrity constraint; assumptions have been extended with starting time for the most recent job and the guard refers to an auxiliary predicate that holds if and only if all three indicated time intervals have a point in common.

```
+printer(lexon2000,S1,F1), +printer(epsmark1993,S2,F2),
   +printer(pewhack2004,S3,F3) ==>
  overlapping((S1,F1),(S2,F2),(S3,F3))  |  fail.
```

We have compared the efficiency of our first implementation of assumptions [12] with the one hardwired into BinProlog for our HYPROLOG print scheduler program (whose complete version can be found in (website URL) for 10 printers and 50 print jobs. The BinProlog version was about 5 times faster. Our present implementation, with specialized predicates for each type of assumption, gave a speedup of 40 percent, or now only 3 times slower than BinProlog.

For abduction, we have compared our system's performance with that of the A-system [20]. The mentioned reference reports a test of an abductive n-queens program that runs very fast in A-system, considerably faster that in our system. However, an inspection of the example shows that the A-system for this program produces quickly one set of constraints which is then solved by a specialized finite-domain solver. It is difficult to translate this example into our system due to the mentioned limitations for negation, which made the program degenerate into a naive generate-and-test algorithm.

In theory, our approach should be superior for programs that involve many resolution steps, and to verify this, we constructed an example updating a database view involving complex joins. The query (update request) in the test is `w(monkey)` where the view is defined as `w(F):- pp(A,B,C,D), qq(C,D,E,F), rr(A,E,F)`. Each of `pp`, `qq`, `rr` provides a link to either a database predicate or an abducible. Integrity constraints express suitable key constraints and the database predicates contains 100, 99, 99 tuples selected carefully so that an immense collection of combinations needs to be tried out before a solution is found. The example is directly translatable between the two systems, and our program in Prolog plus CHR run through the optimizing SICStus Prolog compiler solves the problem in 3 ms (three) whereas A-system spends 6250 ms for the same job; the tests were performed on a 400 MHz Macintosh G4 Powerbook; the programs are available at the HYPROLOG website.

7 Conclusions and Related Work

We have presented HYPROLOG, a new logic programming language which directly and efficiently integrates abducibles and assumptions into Prolog itself, through simply extending it with a few lines of CHR code.

This provides an optimal combination, in which such programs can be written and executed directly, with only a small extra overhead involved when needed. In contrast, known metainterpreter based implementations of abduction incur heavy computational overhead (for instance, [18,20] has the overhead of alternating abductive steps with resolution steps, the latter also simulated by metainterpretation). An important advantage over other known abduction systems is that the full collection of Prolog's built-in facilities (logical as well as impure) are available, including all available libraries and constraint solvers.

We have also described a methodology for implementing HYPROLOG that obtains an execution speed comparable to that of traditional Prolog programs.

The component of a HYPROLOG program that corresponds to a logic program is executed directly as a Prolog program, and its integrity constraints directly as CHR rules. This means that HYPROLOG programs can be run through existing optimizing compilers for Prolog and CHR. It is interesting to point out that integrity constraints are automatically coroutined by virtue of CHR rules.

There are existing, efficient implementations of Assumptive Logic Programs but the present work extends the paradigm with integrity constraints and the option to combine with abduction in a common framework.

The price paid for this efficiency and flexibility is a limitation on the use of negation. Yet even with this restriction, many useful examples are made possible.

Some examples in the literature of abduction involving Event Calculus do not work in our approach but others, such as [24] on robot planning seem possible (this example has been implemented in CHR in an early experiment, but not tested in the present framework). Experiences with HYPROLOG and earlier experiments with similar techniques in CHR indicate a spectrum of interesting programs, and the fact that the paradigm can immediately be combined with any other constraint solvers available in the Prolog version at hand substantiates this viewpoint.

The first observation of the similarity between CHR and abductive logic programming was made by [3] showing that abducible predicates can be represented as constraints in CHR's sense and integrity constraints as rules in CHR. The referenced work describes a translation of a class of abductive logic programs with limited use of negation (similar to the present paper) into $\text{CHR}^{\vee}$ [4] which is an extension to CHR with disjunctions in rule bodies; the main difference is that [3] also translates the logic program component into $\text{CHR}^{\vee}$ so that the efficiency of having Prolog do the resolution steps is lost. CHR based abduction for language processing is applied in the CHRG system [10,9] which is based on bottom-up parsing in CHR.

A proposal for emulating abductive logic programming with assumptions was made in [14]. While less efficient than the present proposal, it allowed the same (abducible) predicate to be either proved normally, if this was possible, or abduced if not. It also put the ability to examine unconsumed assumptions to use in combining for instance defeasible reasoning with abductive logic programming, and in suggesting novel extensions such as conditional abductive logic programming—this latter, by abducing not only predicates, but also clauses.

Abduction by means of CHR has been applied by [11] for natural language grammars with automatic error detection and correction.

As we have noticed, negation is the more complicated part to which we have no solution; [6] sketches an extension of [3]'s method intended for a full use of negation-as-failure in program clauses and integrity constraints; as for [3], no integration with Prolog is provided. Unfortunately, it has not been possible to reconstruct the code from the description in [6] in order to test the method and there appears to be inherent looping problems.

The Demo system described in [8] seems to be the first application of CHR to abduction and similar problems, in the shape of a general metainterpreter for logic programs which is reversible in the sense that it can generate programs to make specified goals provable; this property provided by a constraint solver written in CHR for semantic primitives. In terms of efficiency this system is by no means comparable to what is described in the present paper.

The simultaneous availability of abduction and assumptions facilitates subtler reasoning by making it possible to clearly separate the generation of hypotheses from their confirmation, within a dynamic process where different strategies can be flexibly implemented.

A recent CHR-based system that extends abductive reasoning with the ability to confirm or disconfirm abduced facts (or events, since this system specializes to event-based programming) [1] requires a complex architecture to achieve similar results to ours. This system's hypotheses are, as in our own system, represented by abducibles, but their confirmation or disconfirmation is managed by a specialized proof procedure which can be tuned to be skeptical (i.e, to disconfirm at the end of a computation all hypotheses that remain consistent but have not specifically been confirmed) or credulous (i.e., to confirm all of those).

In HYPROLOG we also represent hypotheses as abduced facts, but their (dis)confirmation proceeds within Prolog's normal proof procedure, and can be done either dynamically, or in a postprocessing stage which can interpret the unconsumed assumptions in a variety of ways, ranging from credulous to skeptical, as needed by the particular application.

Acknowledgements. The authors want to thank Michael Cheng for experimentation with an early version of the methods and for helpful discussions, and Dulce Aguilar-Solis for help with benchmark testing. This work was supported by the CONTROL project, funded by the Danish Natural Science Research Council; we also gratefully acknowledge support from Canada's NSERC Discovery Grant program.

References

1. M.Alberti, F.Chesani, M.Gavanelli, E.Lamma, P.Mello, and P.Torroni. The CHR-based Implementation of a System for Generation and Confirmation of Hypotheses. In *19th Workshop on (Constraint) Logic Programming* Wolf, A., Frühwirth, Th., and Meister, M., (eds.). Ulmer Informatik-Berichte 2005-01, University of Ulm. pp. 111–122. Available at http://www.informatik.uni-ulm.de/epindata/user/11541.218,UIB_2005-01.pdf
2. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 1997.
3. Abdennadher, S. and Christiansen, H., An Experimental CLP Platform for Integrity Constraints and Abduction. In: Proc. of FQAS2000, Flexible Query Answering Systems, Larsen, H.L., Kacprzyk, J., Zadrozny, S., Andreasen, T., and Christiansen, H. (Eds.) pp. 141–152, *Advances in Soft Computing series,* Physica-Verlag (Springer), 2000.
4. Abdennadher, S. and Schütz, H. CHR$^\vee$: A flexible query language. In: Proc. FQAS'98, Flexible Query Answering Systems, Andreasen, T., Christiansen, H., and Larsen, H.L. (Eds.) *Lecture Notes in Artificial Intelligence* 1495, pp. 1–14, Springer, 1998.
5. Betz, H. and Frühwirth, T. A linear logic semantics for Constraint Handling Rules. In: Proc. CP 2005, Eleventh International Conference on Principles and Practice of Constraint Programming. *Lecture Notes in Computer Science*. To appear, 2005.

6. Badea, L. and Tilivea D. Abductive Partial Order Planning with Dependent Fluents. In: KI 2001: Advances in Artificial Intelligence, Joint German/Austrian Conference on AI, Baader, F., Brewka, G., Eiter, T., (eds.) *Lecture Notes in Artificial Intelligence* 2174 p. 63-77, Springer, 2001.
7. Chan, D., Constructive negation based on the database completion, *Proc. of Fifth International Conference and Symposium on Logic Programming*, (eds. Kowalski, Bowen), pp. 111–125, MIT Press, 1988.
8. Christiansen, H. Automated reasoning with a constraint-based metainterpreter, *Journal of Logic Programming*, Vol 37(1–3) Oct–Dec, pp. 213–253, 1998.
9. Christiansen, H., Abductive Language Interpretation as Bottom-up Deduction. In: Natural Language Understanding and Logic Programming, Proceedings of the 2002 workshop, ed. Wintner, S., *Datalogiske Skrifter* vol. 92, Roskilde University, Comp. Sci. Dept., pp. 33–47, 2002.
10. Christiansen, H., CHR grammars. *To appear in International Journal on Theory and Practice of Logic Programming, special issue on Constraint Handling Rules*, 2005.
11. Christiansen, H., and Dahl, V., Logic Grammars for Diagnosis and Repair. *International Journal on Artificial Intelligence Tools*, Vol. 2, no. 3 (2003), pp. 227–248.
12. H. Christiansen and V. Dahl. Assumptions and abduction in Prolog. In S. Muñoz-Hernández, J. M. Gómez-Perez, and P. Hofstedt, editors, *Proceedings of WLPE 2004: 14th Workshop on Logic Programming Environments and MultiCPL 2004: Third Workshop on Multiparadigm Constraint Programming Languages Workshop Proceedings*, pages 87–101, 2004.
13. Christiansen, H., Dahl, V., Meaning in Context, In: Proc. Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05). Dey, A.K., Kokinov, B.N., Leake, D.B., and Turner R.M. (Eds.) *Lecture Notes in Artificial Inteligence* vol. 3554, pp. 97–111, 2005.
14. Dahl, V., and Tarau, P. Assumptive Logic Programming. *Argentine Symposium on Artificial Intelligence (ASAI)* 2004, September 20-21, Cordoba, Argentina, 2004.
15. Dahl, V., Tarau, P., and Li, R., Assumption grammars for processing natural language. *Proc. Fourteenth International Conference on Logic Programming.* Naish, L. (ed.), pp. 256–270, MIT Press, 1997.
16. G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, and C. Holzbaur. Compiling ask constraints. In B. Demoen and V. Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
17. Frühwirth, T.W., Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, Vol. 37(1–3), pp. 95–138, 1998.
18. Kakas, A.C., Michael, A., and Mourlas, C. ACLP: Abductive Constraint Logic Programming, *The Journal of Logic Programming*, vol 44, pp. 129–177, 2000.
19. Kakas, A.C., Kowalski, R.A., and Toni, F. The role of abduction in logic programming, *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pp. 235–324, 1998.
20. Kakas, A.C., Van Nuffelen, B., and Denecker, M. A-System: Problem Solving through Abduction. *IJCAI 2001: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Nebel, B. (ed.), Morgan-Kaufmann, pp. 591–596, 2001.
21. Pereira, F.C.N., and Warren, D.H.D., Definite clause grammars for language analysis. A survey of the formalism and a comparison with augmented transition grammars. *Artificial Intelligence* 10, no. 3–4, pp. 165–176, 1980.

22. Poole, D., Mackworth, A., and Goebel, R. *Computational Intelligence*, Oxford University Press, 1998.
23. *SICStus Prolog user's manual*. Version 3.12, SICS, Swedish Institute of Computer Science, 2005. Most recent version available at `http://www.sics.se/isl`.
24. Shanahan, M., Reinventing Shakey. *Logic-Based Artificial Intelligence*, Minker, J. (ed). Kluwer Academic, pp. 233–253, 1999.
25. Tarau, P., Dahl, V., and Fall, A. Backtrackable State with Linear Affine Implication and Assumption Grammars. In: Concurrency and parallelism, Programming, Networking, and Security, Jaffar, J. and Yap, R. (eds.). *Lecture Notes in Computer Science* 1179, Springer Verlag, pp. 53–64, 1996.