Practical Run-time Checking via Unobtrusive Property Caching

NATALIIA STULOVA[1] JOSÉ F. MORALES[1] MANUEL V. HERMENEGILDO[1,2]

[1] *IMDEA Software Institute*
(*e-mail:* {nataliia.stulova, josef.morales, manuel.hermenegildo}@imdea.org)
[2] *School of Computer Science, Technical University of Madrid (UPM)*
(*e-mail:* manuel.hermenegildo@upm.es)

submitted April 29, 2015; revised July 3, 2015; accepted July 14, 2015

Abstract

The use of annotations, referred to as assertions or contracts, to describe program properties for which run-time tests are to be generated, has become frequent in dynamic programing languages. However, the frameworks proposed to support such run-time testing generally incur high time and/or space overheads over standard program execution. We present an approach for reducing this overhead that is based on the use of memoization to cache intermediate results of check evaluation, avoiding repeated checking of previously verified properties. Compared to approaches that reduce checking frequency, our proposal has the advantage of being exhaustive (i.e., all tests are checked at all points) while still being much more efficient than standard run-time checking. Compared to the limited previous work on memoization, it performs the task without requiring modifications to data structure representation or checking code. While the approach is general and system-independent, we present it for concreteness in the context of the Ciao run-time checking framework, which allows us to provide an operational semantics with checks and caching. We also report on a prototype implementation and provide some experimental results that support that using a relatively small cache leads to significant decreases in run-time checking overhead.

1 Introduction

The use of annotations to describe program properties for which run-time tests are to be generated has become frequent in dynamic programming languages, including assertion-based approaches in (Constraint) Logic Programming ((C)LP) (**?**; Puebla et al. 1997; Bueno et al. 1997; Boye et al. 1997; Hermenegildo et al. 1999; Puebla et al. 2000b; Laï 2000; Hermenegildo et al. 2005; Mera et al. 2009), soft/gradual typing in functional programming (Cartwright and Fagan 1991; Findler and Felleisen 2002; Tobin-Hochstadt and Felleisen 2008; Dimoulas and Felleisen 2011), and contract-based extensions in object-oriented programming (Lamport and Paulson 1999; Fähndrich and Logozzo 2011; Leavens et al. 2007). However, run-time testing in these frameworks can generally incur high penalty in execution time and/or space over the standard program execution without tests. A number of techniques have been proposed to date to reduce this overhead, including simplifying the checks at compile time via static

analysis (Puebla et al. 1997; Bueno et al. 1997; Hermenegildo et al. 1999) or reducing the frequency of checking, including for example testing only at a reduced number of points (Mera et al. 2009; Mera et al. 2011).

Our objective is to develop an approach to run-time testing that is efficient while being minimally obtrusive and remaining exhaustive. We present an approach based on the use of memoization to cache intermediate results of check evaluation in order to avoid repeated checking of previously verified properties over the same data structure. Memoization has of course a long tradition in (C)LP in uses such as tabling resolution (Tamaki and Sato 1986; Dietrich 1987; Warren 1992), including also sharing and memoizing tabled sub-goals (Zhou and Have 2012), for improving termination. Memoization has also been used in program analysis (Warren et al. 1988; Muthukumar and Hermenegildo 1992), where tabling resolution is performed using abstract values. However, in tabling and analysis what is tabled are call-success patterns and in our case the aim is to cache the results of test execution.

While the approach that we propose is general and system-independent, we will present it for concreteness in the context of the Ciao run-time checking framework. The Ciao model (Hermenegildo et al. 1999; Puebla et al. 2000b; Hermenegildo et al. 2005) is well understood, and different aspects of it have been incorporated in popular (C)LP systems, such as Ciao, SWI, and XSB (Hermenegildo et al. 2012; Swift and Warren 2012; Mera and Wielemaker 2013). Using this concrete model allows us to provide an operational semantics of programs with checks and caching, as well as a concrete implementation from which we derive experimental results. We also present a program transformation for implementing the run-time checks that is more efficient than previous proposals (Puebla et al. 2000b; Mera et al. 2009; Mera et al. 2011). Our experimental results provide evidence that using a relatively small cache leads to significant decreases in run-time checking overhead.

2 Preliminaries

Basic notation and standard semantics. We recall some concepts and notation from standard (C)LP theory. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol of arity n and $t_1, ..., t_n$ are terms. A *constraint* is a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H `:-` B where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules.

The *definition* of an atom A in a program, $\mathsf{defn}(A)$, is the set of variable renamings of the program rules s.t. each renaming has A as a head and has distinct new local variables. We assume that all rule heads are *normalized*, i.e., H is of the form $p(X_1, ..., X_n)$ where the $X_1, ..., X_n$ are distinct free variables. Let $\overline{\exists}_L\theta$ be the constraint θ restricted to the variables of the syntactic object L. We denote *constraint entailment* by $\models$, so that $\theta_1 \models \theta_2$ denotes that θ_1 entails θ_2. Then, we say that θ_2 is *weaker* than θ_1.

The operational semantics of a program is given in terms of its "derivations", which are sequences of "reductions" between "states". A *state* $\langle G \mid \theta \rangle$ consists of a goal G and

a constraint store (or *store* for short) θ. We use :: to denote concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. A state $S = \langle L :: G \mid \theta \rangle$ where L is a literal can be *reduced* to a state S' as follows:

1. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ if L is a constraint and $\theta \wedge L$ is satisfiable.
2. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle B :: G \mid \theta \rangle$ if L is an atom of the form $p(t_1, \ldots, t_n)$, for some rule $(L \texttt{:-} B) \in \mathsf{defn}(L)$.

We use $S \rightsquigarrow S'$ to indicate that a reduction can be applied to state S to obtain state S'. Also, $S \rightsquigarrow^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. A *query* is a pair (L, θ), where L is a literal and θ a store, for which the (C)LP system starts a computation from state $\langle L \mid \theta \rangle$. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle \square \mid \theta' \rangle$, where $\square$ denotes the empty goal sequence. In that case, the constraint $\bar{\exists}_L \theta'$ is an *answer* to S. We denote by $\mathsf{answers}(Q)$ the set of answers to a query Q.

`pred`*-Assertions and their Semantics.* Assertions are linguistic constructions for expressing properties of programs. Herein, we will use the `pred`-assertions of (Hermenegildo et al. 1999; Puebla et al. 2000a; Puebla et al. 2000b), for which we follow the formalization of (Stulova et al. 2014). These assertions allow specifying certain conditions on the constraint store that must hold at certain points of program derivations. In particular, they allow stating sets of *preconditions* and *conditional postconditions* for a given predicate. A set of assertions for a predicate is of the form:

$$\begin{array}{l} \texttt{:- pred } Head \texttt{ : } Pre_1 \texttt{ => } Post_1\texttt{.} \\ \ldots \\ \texttt{:- pred } Head \texttt{ : } Pre_n \texttt{ => } Post_n\texttt{.} \end{array}$$

where $Head$ is a normalized atom that denotes the predicate that the assertions apply to, and the Pre_i and $Post_i$ are (DNF) formulas that refer to the variables of $Head$. We assume the Pre_i and $Post_i$ to be DNF formulas of *prop* literals, which specify conditions on the constraint store. A prop literal L *succeeds trivially* for θ in program P, denoted $\theta \Rightarrow_P L$, iff $\exists \theta' \in \mathsf{answers}((L, \theta))$ such that $\theta \models \theta'$.

A set of assertions as above states that in any execution state $\langle Head :: G \mid \theta \rangle$ at least one of the Pre_i conditions should hold, and that, given the $(Pre_i, Post_i)$ pair(s) where Pre_i holds, then, if $Head$ succeeds, the corresponding $Post_i$ should hold upon success. More formally, given a predicate represented by a normalized atom $Head$, and the corresponding set of assertions is $\mathcal{A} = \{A_1 \ldots A_n\}$, with $A_i =$ "$\texttt{:- pred } Head \texttt{ : } Pre_i \texttt{ => } Post_i\texttt{.}$" such assertions are normalized into a set of *assertion conditions* $\{C_0, C_1, \ldots, C_n\}$, with:

$$C_i = \begin{cases} \mathsf{calls}(Head, \bigvee_{j=1}^{n} Pre_j) & i = 0 \\ \mathsf{success}(Head, Pre_i, Post_i) & i = 1..n \end{cases}$$

If there are no assertions associated with $Head$ then the corresponding set of conditions is empty. The set of assertion conditions for a program is the union of the assertion conditions for each of the predicates in the program.

The $\mathsf{calls}(Head, \ldots)$ conditions encode the checks that ensure that the calls to the predicate represented by $Head$ are within those admissible by the set of assertions, and we thus call them the *calls assertion conditions*. The $\mathsf{success}(Head_i, Pre_i, Post_i)$ conditions encode the checks for compliance of the successes for particular sets of calls, and we thus call them the *success assertion conditions*.

We now turn to the operational semantics with assertions, which checks whether assertion conditions hold or not while computing the derivations from a query. In order to keep track of any violated assertion conditions, we add labels to the assertion conditions. Given the atom L_a and the corresponding set of assertion conditions $\mathcal{A}_C$, $\mathcal{A}_C^{\#}(L_a)$ denotes the set of *labeled* assertion condition instances for L_a, where each is of the form $c\#C_a$, such that $\exists C \in \mathcal{A}_C$, $C = \mathsf{calls}(L, Pre)$ (or $C = \mathsf{success}(L, Pre, Post)$), σ is a renaming s.t. $L = \sigma(L_a)$, $C_a = \mathsf{calls}(L_a, \sigma(Pre))$ (or $C_a = \mathsf{success}(L_a, \sigma(Pre), \sigma(Post))$), and c is an identifier that is unique for each C_a. We also introduce an extended program state of the form $\langle G \mid \theta \mid \mathcal{E} \rangle$, where $\mathcal{E}$ denotes the set of identifiers for falsified assertion condition instances. For the sake of readability, we write labels in *negated* form when they appear in the error set. We also extend the set of literals with syntactic objects of the form $\mathsf{check}(L, c)$ where L is a literal and c is an identifier for an assertion condition instance, which we call *check literals*. Thus, a *literal* is now a constraint, an atom or a check literal. We can now recall the notion of *Reductions in Programs with Assertions* from (Stulova et al. 2014), which is our starting point: a state $S = \langle L :: G \mid \theta \mid \mathcal{E} \rangle$, where L is a literal, can be *reduced* to a state S', denoted $S \rightsquigarrow_{\mathcal{A}} S'$, as follows:

1. If L is a constraint or $L = X(t_1, \ldots, t_n)$, then $S' = \langle G' \mid \theta' \mid \mathcal{E} \rangle$ where G' and θ' are obtained in a same manner as in $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G' \mid \theta' \rangle$
2. If L is an atom and $\exists (L \texttt{:-} B) \in \mathsf{defn}(L)$, then $S' = \langle B :: G' \mid \theta \mid \mathcal{E}' \rangle$ where:
$$\mathcal{E}' = \begin{cases} \mathcal{E} \cup \{\bar{c}\} & \text{if } \exists\ c\#\mathsf{calls}(L, Pre) \in \mathcal{A}_C^{\#}(L) \text{ s.t. } \theta \not\Rightarrow_P Pre \\ \mathcal{E} & \text{otherwise} \end{cases}$$
and $G' = \mathsf{check}(L, c_1) :: \ldots :: \mathsf{check}(L, c_n) :: G$ such that $c_i\#\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C^{\#}(L) \ \wedge\ \theta \Rightarrow_P Pre_i$.
3. If L is a check literal $\mathsf{check}(L', c)$, then $S' = \langle G \mid \theta \mid \mathcal{E}' \rangle$ where
$$\mathcal{E}' = \begin{cases} \mathcal{E} \cup \{\bar{c}\} & \text{if } c\#\mathsf{success}(L', \_, Post) \in \mathcal{A}_C^{\#}(L') \wedge \theta \not\Rightarrow_P Post \\ \mathcal{E} & \text{otherwise} \end{cases}$$

3 Run-time Checking with Caching

The standard operational semantics with run-time checking revisited in the previous section has the same potential problems as other approaches which perform exhaustive tests: it can be prohibitively expensive, both in terms of time and memory overhead. For example, checking that the first argument of the `length/2` predicate is a list at each recursive step turns the standard $O(n)$ algorithm into $O(n^2)$.

As mentioned in the introduction, our objective is to develop an effective solution to this problem based on memoizing property checks. An observation that works in our favor is that many of the properties of interest in the checking process (such as, e.g., regtype instantiation checking) are monotonic. That is, we will concentrate on properties such that, for all property checks L if $\langle \_ \mid \theta \rangle \rightsquigarrow^* \langle \_ \mid \theta' \rangle$ and $\theta \Rightarrow_P L$ then $\theta' \Rightarrow_P L$. In this context it clearly seems attractive to keep L in the store so that it does not need to be recomputed. However, memoizing every checked property may also have prohibitive costs in terms of memory overhead. A worst-case scenario would multiply the memory needs by the number of call patterns to properties, which can be large in realistic programs. In addition, looking for stored results in the store obviously also has a cost that must be taken into account.

Operational Semantics with Caching. We base our approach on an operational semantics which modifies the run-time checking to maintain and use a *cache store*. The *cache store* $\mathbb{M}$ is a special constraint store which temporarily holds results from the evaluation of *prop* literals w.r.t. the standard constraint store θ. We introduce an extended program state of the form $\langle G \mid \theta \mid \mathbb{M} \mid \mathcal{E} \rangle$ and a *cached* version of "*succeeds trivially*": given a prop literal L, it succeeds trivially for θ and $\mathbb{M}$ in program P, denoted $\theta \overset{\mathbb{M}}{\Rightarrow}_P L$, iff either $L \in \mathbb{M}$ or $\theta \Rightarrow_P L$. Also, the cache store is updated based on the results of the prop checks, formalized in the following definitions:

Definition 1 (*Updates on the Cache Store*)
Let us consider a DNF formula $Props = \bigvee_{i=1}^{n}(\bigwedge_{j=0}^{m(i)} L_{ij})$, where each L_{ij} is a prop literal. By $\mathsf{lits}(Props) = \{L_{ij} | i \in [1:n], j \in [0:m(i)]\}$ we denote the set of all literals which appear in $Props$. The *cache update* operation is defined as a function $\mathsf{upd}(\theta, \mathbb{M}, Props)$ such that:

$$\mathsf{upd}(\theta, \mathbb{M}, Props) \subseteq \mathbb{M} \cup \{L | (\theta \Rightarrow_P L) \wedge (L \notin \mathbb{M}) \wedge (L \in \mathsf{lits}(Props)\}$$

Note that a precise definition of cache update is left open in this semantics. Contrary to θ, updates to the cache store $\mathbb{M}$ are not monotonic since we allow the cache to "forget" information as it fills up, i.e., we assume from the start that $\mathbb{M}$ is of limited capacity. However, that information can always be recovered via recomputation of property checks. In practice the exact cache behavior depends on parts of the low-level abstract machine state that are not available at this abstraction level. It will be described in detail in later sections.

Definition 2 (*Reductions with Assertions and Cache Store*)
A state $S = \langle L :: G \mid \theta \mid \mathbb{M} \mid \mathcal{E} \rangle$, where L is a literal, can be *reduced* to a state S', denoted $S \rightsquigarrow_{\mathcal{A}} S'$, as follows:

1. If L is a constraint or $L = X(t_1, \ldots, t_n)$, then $S' = \langle G' \mid \theta' \mid \mathbb{M} \mid \mathcal{E} \rangle$ where G' and θ' are obtained in a same manner as in $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G' \mid \theta' \rangle$
2. If L is an atom and $\exists (L \texttt{:-} B) \in \mathsf{defn}(L)$, then
   $S' = \langle B :: G' \mid \theta \mid \mathbb{M}' \mid \mathcal{E}' \rangle$ where:
   $$\mathcal{E}' = \begin{cases} \{\bar{c}\} \cup \mathcal{E} & \text{if } \exists\, c\#\mathsf{calls}(L, Pre) \in \mathcal{A}^{\#}_{C}(L) \text{ s.t. } \theta \overset{\mathbb{M}}{\not\Rightarrow}_P Pre \\ \mathcal{E} & \text{otherwise} \end{cases}$$
   $\mathbb{M}' = \mathsf{upd}(\theta, \mathbb{M}, Pre)$ and $G' = \mathsf{check}(L, c_1) :: \ldots :: \mathsf{check}(L, c_n) :: G$ such that $c_i\#\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}^{\#}_{C}(L) \ \wedge \ \theta \overset{\mathbb{M}}{\Rightarrow}_P Pre_i$.
3. If L is a check literal $\mathsf{check}(L', c)$, then $S' = \langle G \mid \theta \mid \mathbb{M}' \mid \mathcal{E}' \rangle$ where
   $$\mathcal{E}' = \begin{cases} \{\bar{c}\} \cup \mathcal{E} & \text{if } c\#\mathsf{success}(L', \_, Post) \in \mathcal{A}^{\#}_{C}(L') \wedge \theta \overset{\mathbb{M}}{\not\Rightarrow}_P Post \\ \mathcal{E} & \text{otherwise} \end{cases}$$
   and $\mathbb{M}' = \mathsf{upd}(\theta, \mathbb{M}, Post)$.

4 Implementation of Run-time Checking with Caching

We use the traditional definitional transformation (Puebla et al. 2000b) as a basis of our implementation of the operational semantics with cached checks. This consists of a program transformation that introduces *wrapper* predicates that check calls and success

assertion conditions while running on a standard (C)LP system. However, we propose a novel transformation that, in contrast to previous approaches, groups all assertion conditions for the same predicate together to produce optimized checks.

Given a program $\mathcal{P}$, for every predicate p the transformation replaces all clauses $p(\bar{x}) \leftarrow body$ by $p'(\bar{x}) \leftarrow body$, where p' is a new predicate symbol, and inserts the wrapper clauses given by $\mathsf{wrap}(p(\bar{x}), p')$. The wrapper generator is defined as follows:

$$\mathsf{wrap}(p(\bar{x}), p') = \left\{ \begin{array}{l} p(\bar{x}) \text{ :- } p_C(\bar{x}, \bar{r}), p'(\bar{x}), p_S(\bar{x}, \bar{r}). \\ p_C(\bar{x}, \bar{r}) \text{ :- } ChecksC. \\ p_S(\bar{x}, \bar{r}) \text{ :- } ChecksS. \end{array} \right\}$$

where *ChecksC* and *ChecksS* are the optimized compilation of pre- and postconditions $\bigvee_{i=1}^{n} Pre_i$ and $\bigwedge_{i=1}^{n}(Pre_i \rightarrow Post_i)$ respectively, for $c_0\#\mathsf{calls}(p(\bar{x}), \bigvee_{i=1}^{n} Pre_i)$, $c_i\#\mathsf{success}(p(\bar{x}), Pre_i, Post_i) \in \mathcal{A}_C^{\#}(p(\bar{x}))$; and the additional *status* variables $\bar{r}$ are used to communicate the results of each Pre_i evaluation to the corresponding $(Pre_i \rightarrow Post_i)$ check. This way, without any modifications to the literals calling p in the bodies of clauses in $\mathcal{P}$ (and in any other modules that contain calls to p), after the transformation run-time checks will be performed for all these calls to p since p (now p') will be accessed via the wrapper predicate.

The compilation of checks for assertion conditions emits a series of calls to a `reify_check(P,R)` predicate, which accepts as the first argument a property and unifies its second argument with `1` or `0`, depending on whether the property check succeeded or not. The results of those reified checks are then combined and evaluated as boolean algebra expressions using bitwise operations and the Prolog `is/2` predicate. That is, the logical operators $(A \vee B)$, $(A \wedge B)$, and $(A \rightarrow B)$ used in encoding assertion conditions are replaced by their bitwise logic counterparts `R is A \/ B`, `R is A /\ B`, `R is (A # 1) \/ B`, respectively.

The purpose of reification and this compilation scheme is to make it possible to optimize the logic formulae containing properties that result from the combination of several `pred` assertions (i.e., the assertion conditions). The optimization consists in reusing the reified status R when possible, which happens in two ways. First, the *prop* literals which appear in *Pre* or *Post* formulas are only checked once (via `reify_check/2`) and then their reified status R is reused when needed. Second, the reified status of each *Pre* conjunction is reused both in *ChecksC* and *ChecksS*.

In practice the $\mathsf{wrap}(p(\bar{x}),p')$ clause generator shares the minimum number of status variables and omits *trivial* assertion conditions, i.e., those with `true` conditions in one of their parts. For instance, excluding $p_S(\bar{x}, \bar{r})$ preserves low-level optimizations such as last call optimization.[1]

Example 1 (*Program transformation*)
Consider the following annotated program:

```
:- pred p(X,Y) : (int(X) , var(Y)) => (int(X), int(Y)). % A1
:- pred p(X,Y) : (int(X) , var(Y)) => (int(X), atm(Y)). % A2
:- pred p(X,Y) : (atm(X) , var(Y)) => (atm(X), atm(Y)). % A3
```

[1] Even though in this work the $p_C(\bar{x}, \bar{r})$ and $p_S(\bar{x}, \bar{r})$ predicates follow the usual bytecode-based compilation path, note that they have a concrete structure that is amenable to further optimizations (like specialized WAM-level instructions or a dedicated interpreter).

```
p(1,42).  p(2,gamma).  p(a,alpha).
```

From the set of assertions $\{A1, A2, A3\}$ the following assertion conditions are constructed:

$$C_0 = \mathsf{calls}(p(X,Y), (int(X) \wedge var(Y)) \vee ((atm(X) \wedge var(Y))))$$
$$C_1 = \mathsf{success}(p(X,Y), (int(X) \wedge var(Y)), (int(X) \wedge int(Y)))$$
$$C_2 = \mathsf{success}(p(X,Y), (int(X) \wedge var(Y)), (int(X) \wedge atm(Y)))$$
$$C_3 = \mathsf{success}(p(X,Y), (atm(X) \wedge var(Y)), (atm(X) \wedge atm(Y)))$$

The resulting optimized program transformation is:

```
p(X,Y) :-
        p_c(X,Y,R3,R4),
        p'(X,Y),
        p_s(X,Y,R3,R4).

p_c(X,Y,R3,R4) :-
        reify_check(atm(X),R0),
        reify_check(int(X),R1),
        reify_check(var(Y),R2),
        R3 is R1/\R2,
        R4 is R0/\R2,
        Rc is R3\/R4,
        error_if_false(Rc).
```

```
p_s(X,Y,R3,R4) :-
        reify_check(atm(X),R5),
        reify_check(int(X),R6),
        reify_check(atm(Y),R7),
        reify_check(int(Y),R8),
        Rs is (R3#1\/(R6/\R8))
           /\ (R3#1\/(R6/\R7))
           /\ (R4#1\/(R5/\R7)),
        error_if_false(Rs).

p'(1,42).  p'(2,gamma).  p'(a,alpha).
```

Please note that `A1` and `A2` have identical preconditions, and this is reflected in having only one property combination, `R3`, for both of them. The same works for individual properties: in C_0 literal $int(X)$ appears twice, literal $var(Y)$ three times, but all such occurrences correspond to only one check in the code respectively.

The error-reporting predicates `error_if_false/1` in the instrumented code implement the $\mathcal{E}$ update in the operational semantics. These predicates abstract away the details of whether errors produce exceptions, are reported to the user, or are simply recorded.

The cache itself is accessed fundamentally within the `reify_check/2` predicate. Although the concrete details for a particular use case (and a corresponding set of experiments) will be described later, we discuss the main issues and trade-offs involved in cache implementation in this context. First, although the cache will in general be software-defined and dynamically allocated, in any case the aim is to keep it small with a bounded limit (typically a fraction of the stacks), so that it does not have a significant impact on the memory consumption of the program.

Also, in order to ensure efficient lookups and insertions of the cache elements, it may be advantageous not to store the property calls literally but rather their memory representation. This means however that, e.g., for *structure-copying* term representation, a property may appear more than once in the cache for the same term if its representation appears several times in memory.

Furthermore, insertion and removal (*eviction*) of entries can be optimized using heuristics based on the cost of checks (e.g., not caching simple checks like `integer/1`), the entry index number (such as direct-mapped), the history of entry accesses (such as LRU or least-recently used), or caching *contexts* (such as caching depth limits during term traversal in regular type checks).

Finally, failure and some of the stack maintenance operations such as reallocations for stack overflows, garbage collection, or backtracking need updates on the cache entries (due to invalidation or pointer reallocation). Whether it is more optimal to evict some or all entries, or update them is a nontrivial decision that defines another dimension in heuristics.

5 Application to Regular Type Checking

As concrete properties to be used in our experiments we select a simple yet useful subset of the properties than can be used in assertions: the regular types (Dart and Zobel 1992) often used in (C)LP systems. Regular types are properties whose definitions are *regular programs*, defined by a set of clauses, each of the form: "$p(x, v_1, \ldots, v_n)$:- $B_1, \ldots, B_k$" where x is a linear term (whose variables, which are called *term variables*, are unique); the terms x of different clauses do not unify; v_1, ..., v_n are unique variables, which are called *parametric variables*; and each B_i is either $t(z)$ (where z is one of the *term variables* and t is a *regular type expression*) or $q(y, t_1, \ldots, t_m)$ (where $q/(m+1)$ is a *regular type*, $t_1, \ldots, t_m$ are *regular type expressions*, and y is a *term variable*). A *regular type expression* is either a parametric variable or a parametric type functor applied to some of the parametric variables. A parametric type functor is a regular type, defined by a regular program.

Instantiation checks. Intuitively, a prop literal L succeeds trivially if L succeeds for θ without adding new "relevant" constraints to θ (Hermenegildo et al. 1999; Puebla et al. 2000a).[2] A standard technique to check membership on regular types is based on *tree automata*. In particular, the regular types defined above are recognizable by top-down deterministic automata.

This also includes parametric regtypes, provided their parameters are instantiated with concrete types during checking, since then they can be reduced to non-parametric regtypes.

Let us recall some basics on deterministic tree automata, as they will be the basis of our regtype checking algorithm. A tree automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, \Delta, Q_f \rangle$ where Σ, Q, Δ, Q_f are finite sets such that: Σ is a signature, Q is a finite set of states, Δ is the set of transitions of the form $f(q_1, \ldots, q_n) \to q$ where $f \in \Sigma$, $q, q_1, \ldots, q_n \in Q$ with n being the arity of f, and $Q_f \subseteq Q$ is the set of final states. The automaton is *top-down deterministic* if $|Q_f| = 1$ and for all $f \in \Sigma$ and all $q \in Q$ there exists at most one sequence $q_1, \ldots, q_n$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$.

Translation of regular types (or instances of parametric regular types for particular types) from Prolog clauses into deterministic top-down tree automata rules is straightforward. This representation is suitable for low-level encoding (e.g., using integers for q_i states and a map between each q_i state and its definition).

[2] Note that checks are performed via entailment checks w.r.t. primitive (Herbrand) constraints. That means that $term(X)$ (which is always true) and $ground(X)$ (denoting all possible ground terms), despite having the same minimal Herbrand models as predicates, do not have the same s-model and are not interchangeable as regtype instantiation checks.

Algorithm 1 Check that the type of the term stored at x is t, at depth d.

```
function RegCheck(x, t, d)
    Find C ∈ Constructors(t) so that Functor(C) = Functor(x),
    otherwise return False
    if Arity(x) = 0 then                          ▷ Atomic value, not cached
        return True
    else if CacheLookup(x, t) then                ▷ Already in cache
        return True
    else if ∀i ∈ [1, Arity(x)].RegCheck(Arg(i, x), Arg(i, C), d + 1) then
        if d < depthLimit then                    ▷ Insert in cache
            CacheInsert(x, t)
        return True                               ▷ In regtype
    else
        return False                              ▷ Not in regtype
```

Example 2
The following `bintree/2` regular type describes a binary tree of elements of type `T`. The corresponding translation into tree automata rules for the `bintree(int)` instance with $Q_f = \{q_b\}$ is shown to its right.

```
:- regtype bintree/2.
bintree(empty,T).
bintree(tree(LC,X,RC),T) :-
    bintree(LC,T),T(X),bintree(RC,T).
```

$$\Delta = \{ \quad empty \rightarrow q_b \qquad tree(q_b, q_{int}, q_b) \rightarrow q_b \ \}$$

Algorithm for Checking Regular Types with Caches. We describe the RegCheck algorithm for regtype checking using caches in Algorithm 1. The `reify_check/2` predicate acts as the interface between RegCheck and the runtime checking framework. The algorithm is derived from the standard definition of *run* on tree automata. A run of a tree automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, Q_f \rangle$ on a tree $x \in T_\Sigma$ (terms over Σ) is a mapping ρ assigning a state to each occurrence (subterm) of $f(x_1, \ldots, x_n)$ of x such that:

$$f(\rho(x_1), \ldots, \rho(x_n)) \rightarrow \rho(f(x_1, \ldots, x_n)) \in \Delta$$

A term x is recognized by $\mathcal{A}$ if $\rho(x) \in Q_f$. For deterministic top-down recognition, the algorithm starts with the single state in Q_f (which for simplicity, we will use to identify each regtype and its corresponding automata) and follows the rules *backwards*. The tree automata transition rules for a regtype are consulted with the functions $Constructors(t) = \{C | C \rightarrow t \in \Delta\}$, $Arg(i, u)$ (the i-th argument of a constructor or term u), and $Functor(u)$ (the functor symbol, including arity, of a constructor or term u). Once there is a functor match, the regtypes of the arguments are checked recursively. To speed up checks, the cache is consulted (CacheLookup(x, t) searches for (x, t)) before performing costly recursion, and *valid* checks inserted (CacheInsert(x, t) inserts (x, t)) if needed (e.g., using heuristics, explained below). The cache for storing results of regular type checking is implemented as a *set* data structure that can efficiently insert and look up (x, t) pairs, where x is a term address[3] and t a regular type identifier. The specific implementation depends on the cache heuristics, as described below.

[3] Since regtype checks are monotonic, this is safe as long as cache entries are properly invalidated on backtracking, stack movements, and garbage collection. Using addresses is a pragmatic decision to minimize the overheads of caching.

Complexity. It is easy to show that complexity has $O(1)$ best case (if x was cached) and $O(n)$ worst case, with n being the number of tree nodes (or term size). In practice, the caching heuristics can drastically affect performance. For example, assume a full binary tree of n nodes. Caching all nodes at levels multiple of c will need $n/(2^{c+1}-1)$ entries, with a constant cost for the worst case check (at most $2^{c+1}-1$ will be checked, independently of the size of the term).

Cache Implementation and Heuristics. In order to decide what entries are added and what entries are evicted to make room for new entries on cache misses, we have implemented several caching heuristics and their corresponding data structures. Entry eviction is controlled by *replacement policies*:

- Least-recently used (LRU) replacement and fully associative. Implemented as a hash table whose entries are nodes of a doubly linked list. The most recently accessed element is moved to the head and new elements are also added to the head. If cache size exceeds the maximal size allowed, the cache is pruned.
- Direct-mapped cache with collision replacement, with a simple hash function based on modular arithmetic on the term address. This is simpler but less predictable.

The insertion of new entries is controlled by the caching *contexts*, which include the regular type being checked and the location of the check:

- We do not cache simple properties (like primitive type tests, e.g., `integer/1`, etc), where caching is more expensive than recomputing.
- We use the check depth level in the cache interface for recursive regular types. Checks beyond this threshold depth limit are not cached. This gives priority to roots of data structures over internal subterms which may pollute the cache.

Low-level C implementation. In our prototype, this algorithm is implemented in C with some specialized cases (as required for our WAM-based representation of terms, e.g., to deal with atomic terms, list constructors, etc.).[4] The regtype definition is encoded as a map between functors (name and arity) and an array of q states for each argument. For a small number of functors, the map is implemented as an array. Efficient lookup for many functors is achieved using hash maps. Additionally, a number of implicit transition rules exist for primitive types (any term to q_{any}, integers to q_{int}, etc.) that are handled as special cases.

6 Experimental Results and Evaluation

To study the impact of caching on run-time overhead, we have evaluated the run-time checking framework on a set of 7 benchmarks, for regular types. We consider benchmarks where we perform a series of element insertions in a data structure. Benchmarks `amqueue`, `set`, `B-tree`, and (binary) `tree` were adapted from the Ciao libraries;

[4] Even though the algorithm can be easily implemented as a deterministic Prolog program, we chose in this work a specialized, lower-level implementation that can interact more directly with the optimized cache data structures.

Table 1: Benchmarks

benchmark		amqueue		set		AVL-tree		heap		B-tree		RB-tree		tree	
assertions		4		4		8		7		9		15		2	
regtypes		1		1		1		2		5		2		1	
depth limit		2	∞	2	∞	2	∞	2	∞	2	∞	2	∞	2	∞
cache	256	D	D	D	D	L	L	L	L	L	L	L	D	D	L
	128	D	D	D	D	L	L	L	L	D	D	L	D	D	D
size	64	D	D	D	D	L	L	L	L	D	D	L	D	D	D
	32	D	D	D	D	D	D	D	D	D	D	D	D	D	D
max depth	DM	2		1		[7 : 11]		[5 : 11]		[13 : 21]		[6 : 21]		[9 : 20]	
	LRU	2		1		[3 : 11]		[1 : 11]		[4 : 21]		[6 : 20]		[6 : 20]	

benchmarks `AVl-tree`, `RB-tree` and `heap` were adapted from the YAP libraries. These benchmarks can be divided into 4 groups:

(a) simple list-based data structures: `amqueue`, `set`;
(b) balanced tree-based structures that do not change the structural properties of their nodes on balancing: `AVl-tree`, `heap`;
(c) balanced tree-based structures that change node properties: `B-tree` (changes the number of node children), `RB-tree` (changes node color);
(d) unbalanced tree structures (`tree`).

For each run of the benchmark suite the following parameters were varied: cache replacement policy (LRU, direct mapping), cache size (1 to 256 cells), and check depth threshold (1 to 5, and "infinite" threshold for unlimited check depth). Table 1 summarizes the results of the experiments. For each combination of the parameters it reports the optimal caching policy, LRU (L) or direct mapping (D). Also, for each of the benchmarks it reports an interval within which the worst case check depth varies.

The experiments show that the overhead of checks with depth threshold 2 (storing the regtype of the check argument and the regtypes of its arguments) is smaller than or equal to the one obtained with unlimited depth limit (Fig 1). A depth limit of 1 does not allow checks to store enough useful information about terms of most of the data structures (compare the overhead increase for `amqueue` with this and bigger limits), while unlimited checks tend to overwrite this information multiple times, so that it cannot be reused. At the same time, for data structures represented by large nested terms (e.g., nodes of `B-trees`), deeper limits (3 or 4) for small inputs seem more beneficial for capturing such term structure. It can also be observed that the lower cost of element insert/lookup operations with the DM cache replacement policy results in having lower total overhead than with the LRU policy.

While even with caching the cost of the run-time checks still remains significant,[5] caching does reduce overhead by 1-2 orders of magnitude with respect to the cost of run-time checking without caching (Fig. 2). Also, the slowdown ratio of programs with run-time checks using caching is almost constant, in contrast with the linear (or worse)

[5] Note that in general run-time checking is a technique for which non-trivial overhead can be expected for all but the most trivial properties. It can be conceptually associated with running the program in the debugger, which typically also introduces significant cost.

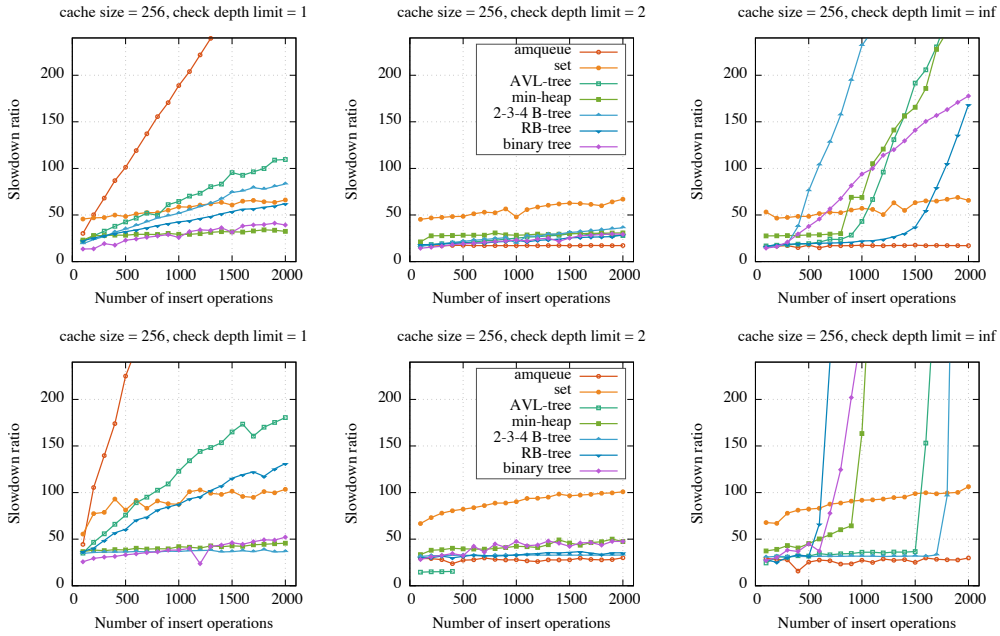


Fig. 1: Run-time check overhead ratios for all benchmarks with check depth thresholds of 1, 2, ∞, and DM (top row) and LRU (bottom row) policies in cache of 256 elements.

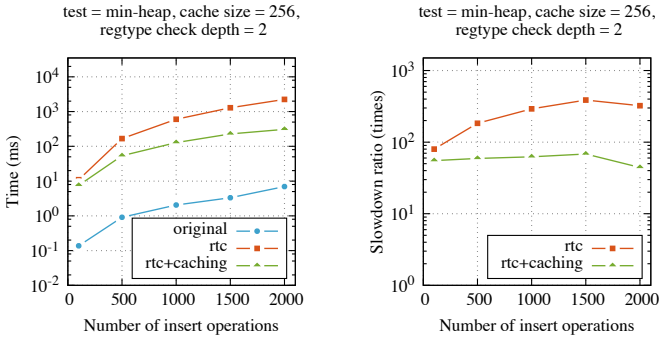
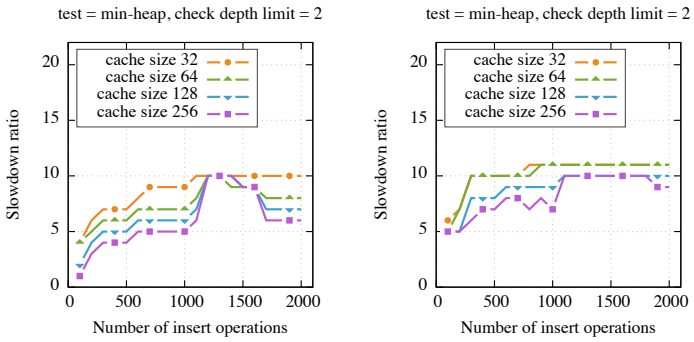


Fig. 2: Absolute and relative running times of the `heap` benchmark with different rtchecks configurations, LRU caching policy.

growth in the case where caching is not used. An important issue that has to be taken into account here is that most of the benchmarks are rather simple, and that performing insert operations is much less costly that performing run-time checks on the arguments of this operation. This explains the observation that checking overhead is the highest for the `set` benchmark (Fig 1), while it is one of the simplest used in the experiments.

Another factor that affects the overhead ratio is cache size. For smaller caches cell rewritings occur more often, and thus the optimal cache replacement policy in such cases is the one with the cheapest operations. For instance, for cache size 32 the optimal policy for all benchmark groups is DM, while for other cache sizes LRU is in some cases better



(a) LRU (b) DM

Fig. 3: Worst case regtype check depth for benchmarks from groups (b) and (c), with LRU and DM cache replacement policies respectively.

as it allows optimizing cell rewritings. This observation is also confirmed by the maximal check depth in the worst case, which is almost half on average for the benchmarks for which LRU is the optimal policy (Fig 3). In the simple data structures of group (a) the experiments show that it is beneficial to have cheaper cache operations (like those of caches with DM caching policy), since such structures do not suffer from cache cell rewritings as much as more complex structures. The same observation is still true for group (d), where for some inputs the binary tree might grow high and regtype checks of leaves will pollute the cache with results of checks for those inner nodes on the path, that are not in the cache, overwriting cache entries with regtypes of previously checked nodes. The DM policy also happens to show better results for group (c) for a similar reason. Since data structures in this group change essential node properties during the tree insertion operation, this in practice means that sub-terms that represent inner tree nodes are (re-)created more often. As a result, with the LRU caching policy the cache would become populated by check results for these recently created nodes, while the DM caching policy would allow preserving (and reusing) some of the previously obtained results. The only group that benefits from LRU is (b), where this policy helps preserving check results for the tree nodes that are closer to the root (and are more frequently accessed) and most of the overwrites happen to cells that store leaves.

More plots are available in the online appendix (Appendix A).

7 Conclusions and Related Work

We have presented an approach to reducing the overhead implied by run-time checking of properties based on the use of memoization to cache intermediate results of check evaluation, avoiding repeated checking of previously verified properties. We have provided an operational semantics with assertion checks and caching and an implementation approach, including a more efficient program transformation than in previous proposals. We have also reported on a prototype implementation and provided experimental results that support that using a relatively small cache leads to very significant decreases

in run-time checking overhead. The idea of using memoization techniques to speed up checks has attracted some attention recently (Koukoutos and Kuncak 2014). Their work (developed independently from ours) is based on adding fields to data structures to store the properties that have been checked already for such structures. In contrast, our approach has the advantage of not requiring any modifications to data structure representation, or to the checking code, program, or core run-time system. Compared to the approaches that reduce checking frequency our proposal has the advantage of being exhaustive (i.e., all tests are checked at all points) while still being much more efficient than standard run-time checking. Our approach greatly reduces the overhead when tests are being performed, while allowing the parts for which testing is turned off to execute at full speed without requiring recompilation. While presented for concreteness in the context of the Ciao run-time checking framework, we argue that the approach is general, and the results should carry over to other programming paradigms.

Acknowledgments: Research supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft*, and Madrid Regional Government S2013/ICE-2731, N-Greens Software.

References

Boye, J., Drabent, W., and Małuszyński, J. 1997. Declarative Diagnosis of Constraint Programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*. U. of Linköping Press, Linköping, Sweden, 123–141.

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszynski, J., and Puebla, G. 1997. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l WS on Automated Debugging–AADEBUG*. U. Linköping Press, 155–170.

Cartwright, R. and Fagan, M. 1991. Soft Typing. In *PLDI'91*. SIGPLAN, ACM, 278–292.

Dart, P. and Zobel, J. 1992. A Regular Type Language for Logic Programs. In *Types in Logic Programming*. MIT Press, 157–187.

Dietrich, S. 1987. Extension Tables for Recursive Query Evaluation. Ph.D. thesis, Departament of Computer Science, State University of New York.

Dimoulas, C. and Felleisen, M. 2011. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst. 33,* 5, 16.

Fähndrich, M. and Logozzo, F. 2011. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*. FoVeOOS'10. Springer-Verlag, Berlin, Heidelberg, 10–30.

Findler, R. B. and Felleisen, M. 2002. Contracts for higher-order functions. In *ICFP*, M. Wand and S. L. P. Jones, Eds. ACM, 48–59.

Hermenegildo, M., Puebla, G., and Bueno, F. 1999. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, Eds. Springer-Verlag, 161–192.

Hermenegildo, M., Puebla, G., Bueno, F., and García, P. L. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., and

Puebla, G. 2012. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming 12,* 1–2 (January), 219–252. http://arxiv.org/abs/1102.5497.

Koukoutos, E. and Kuncak, V. 2014. Checking Data Structure Properties Orders of Magnitude Faster. In *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Lecture Notes in Computer Science, vol. 8734. Springer International Publishing, 263–268.

Laï, C. 2000. Assertions with Constraints for CLP Debugging. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo, and J. Maluszynski, Eds. Lecture Notes in Computer Science, vol. 1870. Springer, 109–120.

Lamport, L. and Paulson, L. C. 1999. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems 21,* 3 (May), 502–526.

Leavens, G. T., Leino, K. R. M., and Müller, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput. 19,* 2, 159–189.

Mera, E., López-García, P., and Hermenegildo, M. 2009. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP'09)*. Number 5649 in LNCS. Springer-Verlag, 281–295.

Mera, E., Trigo, T., López-García, P., and Hermenegildo, M. 2011. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*. LNCS, vol. 6539. Springer-Verlag, 38–53.

Mera, E. and Wielemaker, J. 2013. Porting and refactoring Prolog programs: the PROSYN case study. *TPLP 13,* 4-5-Online-Supplement.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming 13,* 2/3 (July), 315–347.

Puebla, G., Bueno, F., and Hermenegildo, M. 1997. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming.* Available from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz` as technical report CLIP2/97.1.

Puebla, G., Bueno, F., and Hermenegildo, M. 2000a. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming.* Number 1870 in LNCS. Springer-Verlag, 23–61.

Puebla, G., Bueno, F., and Hermenegildo, M. 2000b. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*. Number 1817 in LNCS. Springer-Verlag, 273–292.

Stulova, N., Morales, J. F., and Hermenegildo, M. V. 2014. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *TPLP 12,* 1-2, 157–187.

Tamaki, H. and Sato, M. 1986. OLD Resol. with Tabulation. In *ICLP*. LNCS, 84–98.

Tobin-Hochstadt, S. and Felleisen, M. 2008. The Design and Implementation of Typed Scheme. In *POPL*. ACM, 395–406.

Warren, D. S. 1992. Memoing for Logic Programs. *CACM 35,* 3, 93–111.

Warren, R., Hermenegildo, M., and Debray, S. K. 1988. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*. MIT Press, 684–699.

Zhou, N.-F. and Have, C. T. 2012. Efficient Tabling of Structured Data with Enhanced Hash-Consing. *Theory and Practice of Logic Programming 12*, 547–563.