A Feasibility Study
The Succinct Solver v2.0, XSB Prolog v2.6, and Flow-Logic Based Program Analysis for Carmel *

Authors	: Henrik Pilegaard (hepi@imm.dtu.dk)
Date	: September 30, 2003
Number	: SECSAFE-IMM-008-1.0
Classification	: Internal

Abstract. We perform a direct comparison of the Succinct Solver v2.0 and XSB Prolog v2.6 based on experiments with Control Flow Analyses of scalable Discretionary Ambient programs and Carmel programs. To facilitate this comparison we expand ALFP clauses accepted by the Succinct Solver into more general Normal clauses accepted by both solvers and run the experiments for all three possible combinations of input and solver. This allows the solvers to be tested on even ground and enables the reuse of existing analyses and their corresponding ALFP constraint generators. The performance of the Succinct Solver is at worst a small constant factor worse than XSB Prolog. In optimum cases the Succinct Solver outperforms XSB Prolog by having a substantially lower asymptotic complexity.

Contents

1 Introduction **3**

2 Solver Technologies **5**
- 2.1 The Succinct Solver . . . 5
- 2.2 XSB Prolog . . . 6
- 2.3 Translating ALFP Programs to Normal Programs . . . 9

3 Test Setup and Procedure **12**
- 3.1 Investigated Factors . . . 12
- 3.2 Test Suite . . . 14
- 3.3 Timing the Experiments . . . 14
- 3.4 Presenting the Results . . . 14

4 Discretionary Ambients: Scalable Router Programs **16**
- 4.1 The Language . . . 16
- 4.2 The Analyses . . . 16
- 4.3 Representation of Constraints . . . 17
- 4.4 Test Programs . . . 18
- 4.5 Test Results . . . 19

*Supported in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies, under the IST-1999-29075 project SecSafe; and the Danish SNF-project LoST 21-02-0507.

4.6 Discussion 23

5 Carmel: Demoney - The Electronic Java Card Purse **25**
5.1 The Language 25
5.2 The Analysis 25
5.3 Constraints and Translation 25
5.4 Test Programs 26
5.5 Test Results 27
5.6 Discussion 32

6 Conclusion **33**

A 0-CFA for the 1(m) Programs **37**
A.1 Individual Solvers 37
A.2 Solvers Compared 38

B 0-CFA for the s(m) Programs **39**
B.1 Schematic Drawing 39
B.2 Problem Instance Code for $m = 2$ 39
B.3 Individual Solvers 40
B.4 Solvers Compared 41

C 0-CFA for the lvg(m) Programs **42**
C.1 Individual Solvers 42
C.2 Solvers Compared 43

D 0-CFA for the sph(m) Programs **44**
D.1 Individual Solvers 44
D.2 Solvers Compared 45

E Closure Condition Examples **46**
E.1 da0_2 46
E.2 da0_7 47
E.3 da0_7 - Horn Expanded 48

F Carmel XSB Runs: demoney **49**

G Carmel XSB Runs: demoney-local **50**

H Carmel XSB Runs: jc212api **52**

I Carmel XSB Runs: API **54**

J Carmel XSB Runs: demoney-impl-api **56**

1 Introduction

Static analysis of programs is one of the formal methods for program verification that enjoys widespread use. Formally the approach lies somewhere between automated theorem proving and model checking incorporating the best of both worlds.

Analyses are often constructed in a two-phase process. The first phase concerns the extraction of sets of constraints from programs and the second the solving of these sets.

A benefit of this approach is that solver technology may be shared among applications in a variety of programming languages and among many classes of problems. This emphasizes the fact that the solver technology is usually, if not always, the limiting factor and, thus, that it is important to examine and compare different solver technologies in order to minimize the limitations imposed by the choice of technology.

As the use of static analysis for verification of *Java Card* programs is the corner stone of the *SecSafe* project, it is clearly relevant to identify and evaluate appropriate technologies. Thus, this document in part constitutes a comparison of v2.0 of the *Succinct Solver* (Succinct Solver) of Nielson and Seidl [10] to v2.6 of the general purpose tabled Prolog system *XSB* [15][13], with respect to their usability for Flow-logic based static analysis of programs.

The algorithmic foundation of the Succinct Solver was designed to give state-of-the-art asymptotic worst-case performance while allowing for even better performance in benign cases. This was a conscious decision motivated by the belief that static analysis generally gives rise to problems where the worst-case complexity is not inherent. The resulting solver implementation embodies a functional implementation (in *New Jersey SML*) of an algorithm that computes *stable models* of formulas from the alternation-free fragment of *Least Fix-point Logic* (ALFP formulas) with the same complexity with which they are checked.

The other investigated solver, XSB Prolog, has not to our knowledge been designed with similar considerations in mind. Still, the XSB Prolog system is known to constitute a highly optimized low-level implementation of an algorithm that aims to exhibit good data complexity when computing *well-founded* models for *General Logic Programs*. Given these characteristics one cannot help wondering if (as it has often been suggested to us) this computation engine could be a more efficient alternative to the Succinct Solver - even for benign problems arising in static analysis of programs.

Another concern is the feasibility of the static analysis approach to Java Card security. The elements necessary in order to investigate this, at least from a technology scalability perspective, have long been established within the project. A rational reconstruction suitable for formal verification, *Carmel* (syntax [6], semantics [16]), of the *Java Card Virtual Machine Language* has been established as the language of study. Furthermore, a suitable case study of industrial proportions, *Demoney - The Electronic Purse Applet* [8][7], has been developed by the industrial partner *Trusted Logic* and supplied along with a suitable partial implementation of the *Java Card API v.2.12*. Finally, realistic *control flow analyses* have been defined, e.g. in [4] [3], and implemented, e.g. in [5], in order to be used for feasibility studies and benchmarking.

This document serves a double purpose. We compare the Succinct Solver and XSB Prolog experimentally in order to answer some of the questions raised with respect to their relative efficiency. We also conduct a feasibility study regarding the use of static analysis for Java Card programs. As part of the investigation we conduct a number of experiments applying a 0-CFA analysis to scalable Discretionary Ambient programs in a manner similar to that described in [1]. Furthermore, we combine the

mentioned elements from the SecSafe repository in order to conduct the feasibility study experiments. These experiments also provide valuable insights with respect to the comparative performance and *SecSafe* relevance of the two solvers.

In section 2 we introduce the basics of the investigated solver technologies with the emphasis on XSB Prolog, which is the new system in SecSafe. In section 3 we go on to outline the experimental setup. In section 4 we report on a set of experiments based on scalable ambient programs. These experiments aim to compare the performance of the solver algorithms when subjected to problems of various difficulty. Finally, in section 5 we report on experiments based on a set of Carmel programs that are all connected to the *Demoney* case study.

2 Solver Technologies

In the following the two competing solver technologies will be introduced. Since the Succinct Solver has already been discussed in conjunction with SecSafe a number of times there will be an emphasis on describing the relevant aspects of the XSB system.

Furthermore, we will discuss a general translation - from the more expressive alternation-free Least Fixed Point (ALFP) fragment of First Order Logic (FOL) used in the Succinct Solver to a more restrictive Normal clause form[1] usable in both the Succinct Solver and XSB; this allows us to test the solvers on 'even ground' while reusing constraint generators targeting ALFP Logic.

2.1 The Succinct Solver

The Succinct Solver is a fixed point computation engine able to compute the *stable model* of the *alternation-free Least Fixed Point Logic* [10] - the abstract syntax of which is shown in figure 1.

ALFP formulas extend Horn clauses (with explicit quantification) as they additionally allow both existential quantification in preconditions, negated queries, disjunctions of preconditions, and conjunctions of conclusions. Also, ALFP formulas are subject to a strict syntactical left-to-right stratification requirement (as known from Datalog) that allows negation to be dealt with in a convenient manner. Such stratified *Least Fixpoint formulas formulas* are *alternation-free*. When a predicate symbol

$$
\begin{array}{lcl}
t & ::= & a \mid x \mid f(t_1, \ldots, t_k) \\
\\
pre & ::= & R(t_1, \ldots, t_k) \mid \neg R(t_1, \ldots, t_k) \mid pre_1 \wedge pre_2 \\
 & \mid & pre_1 \vee pre_2 \mid \exists x : pre \mid t_1 = t_2 \mid t_1 \neq t_2 \\
 & \mid & \mathbf{1} \mid \mathbf{0} \\
cl & ::= & R(t_1, \ldots, t_k) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \\
 & \mid & pre \Rightarrow cl \mid \forall x : cl
\end{array}
$$

Figure 1: The ALFP Logic of the Succinct Solver

$R(\ldots)$ occurs in pre-conditions we call it a *query*; and similarly we call $\neg R(\ldots)$ a *negative query*. When $R(\ldots)$ occurs in conclusions we call it an *assertion* (of predicate R).

Clauses in ALFP Logic naturally arise in the context of static program analysis and in some respects the Succinct Solver is a highly specialized tool. The disciplined use of continuations and memoization allows for a very succinct *functional* specification of the solver algorithm [10] and the Succinct Solver (v2.0) [17] used here is an SML implementation of this continuation passing, memoizing algorithm. Contrary to the v1.0 implementation used in [1] the v2.0 implementation allows fixed points to be computed over *unbounded* finite universes, which makes the Succinct Solver and XSB Prolog[2] more even in this respect.

The optimal solution ρ exceeding an interpretation ρ_0 of any ALFP formula cl may always be computed with the asymptotic worst-case complexity

$$\mathcal{O}(|\rho| + |\mathcal{U}|^r \cdot n)$$

[1] Similar to Horn clause form, but allowing negative literals as sub-goals.
[2] Which computes over finite subsets of potentially infinite Herbrand Universes.

where $|\mathcal{U}|$ is the size of the universe, n is the size of cl, r is the maximal nesting depth of quantifiers in cl, and $|\rho|$ is the sum of cardinalities of predicates $\rho(R)$. However, the algorithm of the solver is designed to perform significantly better than this worst-case bound whenever possible, i.e. in benign cases.

We will use the order of the relationship between the size of the solution $|\rho|$ and the size of the universe $|\mathcal{U}|$

$$\mathcal{O}(\frac{|\rho|}{|\mathcal{U}|})$$

as an *analysis population measure* in order to distinguish benign cases. This measure is generally on the order of

$$\mathcal{O}(\sqrt[\kappa]{|\mathcal{U}|})$$

for some value of the *population parameter* $\kappa \in \mathbb{R}$. If $\kappa = 0$ we say that the analysis is lightly populated, otherwise we say it is heavily populated. Given the design of the Succinct Solver we expect it to perform substantially better for lightly populated analyses.

2.2 XSB Prolog

XSB (eXtended Stony Brook) Prolog is a general purpose logic programming system that extends the features of Prolog with tabling, higher order constructs (HiLog), and dynamic code assertion with flexible indexing on dynamically asserted predicates. Programs containing negation are evaluated according to the well-founded semantics and, thus, XSB is able to compute the *Well-Founded model* of General Logic programs.

As a result of these extensions, XSB is generally more useful, more flexible, and more efficient than implementations of standard Prolog. The focus of the present work is on the use of XSB as a solver for constraints that arise in flow-logic based program analysis, and the features of the system will be discussed in this context.

2.2.1 Input Format and HiLog

The syntax of XSB Prolog is that of HiLog [2]. The fragment of this syntax relevant to the present work is shown in figure 2.

$$\begin{array}{lcl} t & ::= & a \mid x \mid t(t_1, \ldots, t_k) \\ l & ::= & t \mid \neg t \mid t_1 = t_2 \mid t_1 \neq t_2 \\ c & ::= & t \mid t \longleftarrow l_1, \ldots, l_k \\ p & ::= & c\,.\ p \mid c. \end{array}$$

Figure 2: The Base Syntax of XSB Prolog

As shown in figure 2, *terms*, t, are either *variables*, x, *constants* (or *atomic symbols*), a, or *structured entities* with a *functor* t and *arguments* t_i. *Literals*, l, are either terms, t, *negation of terms*, $\neg t$, *explicit term unifications*, $t_1 = t_2$, or the opposite, $t_1 \neq t_2$. *Clauses*, c, are then either grounds *facts*, t, or *rules* - each with a *head*, t (the conclusion), and a *body* consisting of a conjunction of literal *subgoals*, l_i (the premises). Finally, *Programs*, p, consist of a conjunction of clauses, c.

We see that the XSB program syntax is basically that of definite programs[3] extended with negation - also called Normal Logic Programs. So, on the surface the syntax of XSB seems similar to that of ordinary Prolog. In reality, however, predicate and relation names, t can be structured or variable as the HiLog syntax allows terms as functors. Still, while XSB has a higher order syntax, it maintains a clean declarative first order semantics allowing full compilation of HiLog predicates via a first order encoding.

Furthermore, a number of operators and constructs not described in figure 2 are available within the system. Thus arithmetic expressions, conditional expressions, and various extra-logical constructs such as *cut* are expressible in XSB.

For the purpose of this paper we do not go beyond the outlined (Prolog) subset of the XSB syntax as it is fully sufficient to express constraints as Normal programs.

2.2.2 Dynamic Code and Indexing

XSB contains a variety of features to support in-memory data-oriented applications. A variety of read and assert mechanisms allow dynamic predicates to be loaded and asserted on the fly and in a backtrackable manner.

Indexing of dynamically loaded code is particularly flexible. In this case indexes can be created on alternate or joint arguments. Also, indexing can be hash- or trie-based if it is necessary to perform indexing deep within a term.

For static code XSB provides unification factoring (also called Transformational Indexing [15]) which extends clause indexing by factoring common unifications out of clause heads. Indexing of static code, though, can only created on argument numbers of predicates.

In order to separate the preprocessing and solving stages as much as possible the tests conducted here rely on statically compiled programs. Furthermore - as tabling is used pervasively, providing efficient trie based indexing - argument (re-)indexing beyond the default indexing of the first argument of predicates is not used.

2.2.3 Tabling and Evaluation Strategies

The XSB system uses SLG resolution (Linear resolution for General sentences with Selection function) as its basic evaluation strategy. SLG extends OLDT resolution (Ordered Linear resolution for Definite sentences with Tabling) with evaluation according to the Well-Founded Semantics. XSB is thus able to employ tabling even for programs containing non-stratified negation. When tabling is not required SLDNF resolution (SLD + Negation as Failure) is used as an optimization.

The use of tabling (also called memoization, lemmatization, caching, or well-formed substring tabling) avoids redundant computation by remembering subcomputations. These are stored as nested sets of terms maintained as tries. Their answers are then reused to respond to later identical requests. This ensures that distinct computations are only performed once and that recursive non-terminating computations are broken. Thus, tabling can be used both to obtain termination of a larger class of recursive programs as well as more efficient memory and CPU usage.

XSB offers a number of options for the use of tabling. For each predicate in a program the programmer may explicitly decide the desired tabling mode. The default option is to use no tabling, but two other options exist.

[3] Conjunctions of definite (positive Horn) clauses.

Variant Tabling only remembers a new subcomputation if it is not a variant, i.e. identical up to α-conversion of variables, of an already computed one. Similarly a new answer to a subcomputation is only remembered if it is not a variant of an existing one. This tabling method interacts well with the *Well-Founded Semantics* and the extra-logical constructs of Prolog.

Subsumptive Tabling only remembers a new subcomputation if it is not subsumed[4] by an already computed one. Similarly a new answer to a subcomputation is only remembered if not subsumed by an existing one. Due to greater reuse this tabling method is superior in terms of space and time efficiency. There are, however, also shortcomings to this tabling method:

- It does not interact well with some of the extra-logical constructs of Prolog, with which variant tabling does.
- Subcomputations involving subsumptive predicates are required to be LRD (Left-to-Right Dynamic) stratified.
- In XSB v2.6 subsumptive tabling does not support the Well-Founded Semantics.

In the present work we are interested in examination of the entire state-space and it makes most sense to table pervasively - i.e. to table all relations. We also want to optimize the time and space behavior of the solving process and therefore subsumptive tabling is the tabling method of choice.

2.2.4 Memory Management, Scheduling, etc.

The memory management strategy of XSB v2.6, SLG-WAM (SLG + Warren Abstract Machine), shares stack elements between computation states. This is fast and memory efficient for tabled evaluation.

There are two possible scheduling strategies that may be employed for tabled evaluation. The default strategy, *Local Evaluation*, ensures that, whenever possible, subgoals are fully evaluated before their answers are returned. This potentially offers superior performance for tabled negation. The alternative strategy, *Batched Evaluation*, ensures that answers are returned as quickly as possible when subgoals are evaluated. This scheduling strategy consistently offers the best performance with respect to the constraint solving that is the subject of this document.

2.2.5 XSB Test Configuration

As evident from the above description a substantial number of parameters may affect the performance of the XSB system when used as a constraint set solver.

For the XSB system to perform as well as possible on the conducted tests we use the system settings shown in table 1.

[4]For one query, $\mathsf{friend}(X, max)$, to be *subsumed* by another, $\mathsf{friend}(X, Y)$, means means that the first is obtainable by α-conversion of the other, but not necessarily the other way around. An ordering which also implies an inclusion ordering of the corresponding answer sets. $\mathsf{ans}(\mathsf{friend}(X, max)) \subseteq \mathsf{ans}(\mathsf{friend}(X, Y))$.

Syntax	Prolog (First Order fragment)
Evaluation strategy	Batched
Memory management	SLG-WAM
Garbage collection	Indirection
Tabling	Subsumptive
Tabling heuristic	Table all
Indexing	Default (1st argument)
Code type	Statically Compiled
Predicate specialization	Off
Unification Factoring	Off

Table 1: Initial settings of XSB

2.3 Translating ALFP Programs to Normal Programs

In order to test the solvers on essentially the same input we define a translation from ALFP clauses to *Normal programs* [5]. This process, which is a straightforward *Horn expansion*, we apply to all generated ALFP constraints before emitting them to either the Succinct Solver or XSB.

In this process, the expansion step, which operates directly on data stored in the *HornDirect* format of the Succinct Solver [17], is shared between the two solvers. For the Succinct Solver a subsequent emission step is not required as the HornDirect data can be loaded directly. For XSB, on the other hand, an emission step performing additional subtle rewriting is required. Both the expansion step and the XSB-specific emission step will be explained in the following.

2.3.1 Horn Expansion of Formulas

Since Prolog systems do not use explicit quantifiers, but *implicitly* consider all clauses universally quantified, we have to deal with the quantifiers. Due to the fact that ALFP Logic allows the existential quantifier to range over premises but not entire clauses we have:

$$(\exists x : P(x)) \Rightarrow \phi \quad \Leftrightarrow \quad \forall x : P(x) \Rightarrow \phi \qquad \text{iff} \qquad x \notin \mathsf{fn}(\phi)$$

Also, the scope of universal quantification may be extended:

$$\phi \Rightarrow \forall x : P(x) \quad \Leftrightarrow \quad \forall x : \phi \Rightarrow P(x) \qquad \text{iff} \qquad x \notin \mathsf{fn}(\phi)$$

Both of these rewriting rules are subject to syntactical restrictions. Conflicts, however, can always be (and are) resolved by renaming the bound variable.

Whenever conjunctions are present in the conclusion (or, equivalently, disjunctions in the premise) of a clause it exceeds the expressiveness of Horn Logic and must be simplified. Conjunctive conclusions are taken care of by:

$$\phi \Rightarrow \omega \wedge \psi \quad \Leftrightarrow \quad (\phi \Rightarrow \omega) \wedge (\phi \Rightarrow \psi)$$

And disjunctive premises by:

$$\phi \vee \omega \Rightarrow \psi \quad \Leftrightarrow \quad (\phi \Rightarrow \psi) \wedge (\omega \Rightarrow \psi)$$

[5] Definite programs (Horn clause programs) extended with default negation.

Note that these two rules duplicate code.

Finally, ALFP Logic allows implications to be nested to arbitrary depth. Again, this is beyond the capabilities of Horn Logic and we apply the following rule:

$$\phi \Rightarrow \omega \Rightarrow \psi \quad \Leftrightarrow \quad \phi \wedge \omega \Rightarrow \psi$$

The actual implementation of this translation applies these rewriting rules in the order of description. Each of the rules regarding quantifiers are applied as one-pass functions while the remaining three functions are applied iteratively.

The result of such a translation can be seen in appendix E.3, which shows the Horn expanded version of the closure condition $\mathsf{da0\_7}$[6] found in appendix E.2.

2.3.2 Emission of XSB code

Besides the general syntactical issues there are only two particular cases left to be considered after the translation step.

Handling Negative Queries As previously mentioned (section 2.1) ALFP Logic imposes very strict stratification requirements on formulas. Obviously the Horn expansion of ALFP formulas gives rise to Normal programs that are also strongly stratified in a global Left-to-Right sense.

The XSB system also imposes some stratification requirements:

- Evaluation of untabled programs (or subprograms) is subject to the classical Prolog stratification requirement [12].
- Evaluation of tabled programs only involving variant tabled predicates does not require stratification - the well-founded semantics allows partial results [13].
- Due to a weakness of subsumptive tabling (section 2.2.3) sub-computations in which subsumptively tabled predicates take part are subject to a requirement of *LRD-stratification* [13].

All of these requirements are substantially weaker than those of ALFP Logic and it is always possible to translate negative queries directly.

Unfortunately XSB differs syntactically between negation of tabled and untabled predicates. Table 2 shows the translation associated with the different tabling modes.

ALFP	XSB no tabling	XSB variant	XSB subsumptively
$\neg R(t_1, \ldots, t_n)$	$\backslash+\ R(t_1, \ldots, t_n)$	$\mathsf{tnot}(R(t_1, \ldots, t_n))$	$\mathsf{tnot}(R(t_1, \ldots, t_n))$

Table 2: Translation of negation for different tabling modes

Handling Explicit Equality and Inequality In the Succinct Solver clauses are always evaluated in the context of a finite universe. Therefore, clauses with a leading explicit check for *inequality* of terms, such as

$$\forall x.\ \text{john} \neq x \wedge \mathsf{friend}(\text{max}, x) \Rightarrow \mathsf{friend}(\text{jim}, x),$$

[6] Embodies one of the analyses we will later use for Discretionary Ambients in section 4.

can always be meaningfully evaluated. In XSB, which does not operate with a similar notion of finite universes, assertions of this type constitute checks that succeeds if the terms are not unifiable. Obviously such a check always fails if conducted before a value has been assigned to x. When translating we solve this problem by reordering the preconditions such that x is always assigned a value before the unification property is tested:

$$\forall x.\ \mathsf{friend}(\max, x) \wedge \mathrm{john} \neq x \Rightarrow \mathsf{friend}(\mathrm{jim}, x)$$

When the precondition of a clause contains an explicit check for *equality* of terms the issue becomes more complicated. As for inequality checks the Succinct Solver simply interprets such assertions as further restrictions on the already finite universe. XSB, on the other hand, interprets the assertion as a request for explicit unification. Therefore, clauses like

$$\forall i.\forall j.\forall x.j = \mathsf{suc}(i) \wedge R(x, i) \Rightarrow R(x, j),$$

cause non-termination in XSB as the number of possible unifications is potentially infinite. When translating we solve this by pre-computing a specialized universe[7], e.g. of numbers num, from which j can be chosen meaningfully.

$$\forall i.\forall j.\forall x.\mathsf{num}(j) \wedge j = \mathsf{suc}(i) \wedge R(x, i) \Rightarrow R(x, j)$$

Getting the Syntax Right When emitting the program we use the above guidelines regarding negative queries and explicit checks for equality/inequality and the following general conventions regarding syntax:

1. Quantifiers are removed completely as they are implicit in XSB.
2. The first letter of variable names is capitalized and the first letter of constant names is changed to lower case in order to suit the syntactical conventions of XSB.
3. Clauses are "reversed" such that conclusions become heads and premises become bodies.

[7] Constructed to faithfully represent the corresponding part of the Succinct Solver universe.

3 Test Setup and Procedure

In order to fairly compare the Succinct Solver v2.0 and XSB v2.6 a number of choices have been made. The following section serves to clarify the purposes, contents, and procedures of the tests performed.

3.1 Investigated Factors

When considering the performance of a logic based solver a number of factors may prove to be relevant. Previous work has established that a minimum of three factors affect the performance of the Succinct Solver:

1. The order of conjuncts in preconditions
2. The use of memoization
3. The order of parameters of relations

XSB is somewhat different. Some of the above factors remain important, some of them less so, and some important factors are new. In the following we will discuss the involved issues in order of importance.

3.1.1 The Order of Conjuncts in Preconditions

For the Succinct Solver it has been shown that the order of the conjuncts in preconditions has a potentially large impact on performance. Due to the left-to-right evaluation efficient search-space reductions may be obtained by reorderings that shift the most restrictive (most likely to fail) subgoals to the front (left).

XSB has the ability to make a choice of different rules for each subgoal when evaluating clauses. This makes the evaluation order a little less static, but individual clauses are still evaluated in a left-to-right manner. As a result XSB may be expected to behave similarly to the Succinct Solver with respect to precondition conjunct ordering.

3.1.2 The Order of Queries

Contrary to the Succinct Solver, XSB, which is a Prolog system, does not automatically solve for the entire solution space when executing a program. Hence, complete solutions for given predicates have to be queried for explicitly - via special predicates issued either as part of the program or manually. As tables are kept in memory until explicitly deleted it seems fair to assume that the order of queries to predicates is important.

3.1.3 The Use of Memoization, Tiling, and Predicate Specialization

For each of the two solvers a number of techniques may be employed in order to prevent variable environments to be propagated or choice points to be laid out unnecessarily.

Memoization serves to prevent propagation of identical environments through existential or disjunctive preconditions. The previous study showed that a syntactic transformation, replacing universal quantification with existential quantification whenever possible, had a positive effect on performance.

In XSB predicates are tabled to improve performance and/or termination characteristics of programs. Evidently, due to the form of the constraints, pervasive tabling is necessary for termination and efficiency when using XSB as a constraint solver for static analysis of programs..

For the comparison to have merit we need to test the solvers on identical input. In the process of translation we eliminate all existential quantifications and all disjunctive preconditions in order to cater for the syntax of XSB. This step is necessary - but as a result the Succinct Solver uses no memoization while XSB tables everything. Thus, it is not possible to evaluate the comparative effects of tabling in the two solvers if insisting on identical input.

Tiling aims to prevent unnecessary propagation of environments by reducing quantifier nesting depth as much as possible. The technique is based on a source level program transformation that introduces auxiliary relations whenever a variable, x, involved in precondition queries does not appear in the conclusion of a clause. Tiling was presented in [9], investigated in [1], and we refer to these places for details.

XSB evaluates queries to programs given in Datalog with negation with a data complexity of $\mathcal{O}(k^v)$ [18], where k is the number of constants in the Herbrand base (i.e. in the program) and v is the maximum number of distinct (and implicitly universally quantified) variables in any clause of the program. This bound implies that tiling, i.e. reducing this maximum as much as possible, would be as beneficial for the complexity of XSB evaluation as it is for the complexity of Succinct Solver evaluation.

As tiling is a solver-external technique and, thus, not crucial to the direct comparison undertaken here we do not investigate this claim any further in this document.

Predicate Specialization is an XSB compile-time source level transformation of a program to a residual program. In the residual program partially instantiated calls to predicates in the original program are replaced with calls to specialized versions of these predicates. While this specialization is expected to result in more efficient execution, preliminary testing has shown no such effect.

Thus, predicate specialization has not been included in the present investigation.

3.1.4 Parameter Ordering and Indexing

When using tabling in XSB the results of subgoal queries are stored in nested tries corresponding to the prefix trees of the Succinct Solver. Thus, the indiscriminate use of tabling, which appears to be appropriate for constraint solving, provides an effective means of indexing in itself. Indeed it renders the effects of static re-indexing and unification factoring (transformational indexing) void.

In the Succinct Solver the indexing is decided during preprocessing. The previous study [1] showed that the indexing procedure works well, in the sense that we cannot reorder the relation parameters of the constraints in a way that gives a better result. [8]

Due to these facts we do not investigate indexing and parameter reordering any further in this paper.

[8] As the positive effect relies on multiple indexes for predicates it may be costly in terms of memory, however.

3.2 Test Suite

We include benchmarks based on discretionary ambients as previously used in [1] to obtain the following advantages:

- We can easily generate and compare test samples of required sizes and complexities.
- The analysis and the corresponding constraint generators are given.
- The testing methodology is fixed in advance.
- We have apriori knowledge of the results to expect.

We also include quantitative results regarding Carmel programs. Specifically we aim to compare analyses of the Carmel application *Demoney* developed by Trusted Logic as a showcase of industrial proportions. By doing this we obtain:

- Results showing how the order of queries affect the performance of XSB.
- Feedback regarding the analysis and the constraint generator for Carmel.
- An indication of the comparative SecSafe relevance of the two solvers.
- Information regarding the feasibility of the static analysis approach to Java Card program verification.

3.3 Timing the Experiments

For the experiments we adopt the timing methodology described in [1].

For the Succinct Solver this means that the procedure is split into two distinct steps. In the initialization step the constraints are loaded into memory and all internal structures are initialized. In the solve step the actual analysis result is then computed. These two steps are timed separately and all times are collected both with and without the time specifically used for garbage collection.

For XSB a similar division is achieved by the use of pre-compiled programs. The Horn expanded Normal program is output in XSB format to a file that is subsequently compiled. This compilation corresponds to the initialization step of the Succinct Solver and performs all of the structure initializations. The subsequent computation of the analysis from the compiled program corresponds to the solve step of the Succinct Solver. Again, we collect the times with and without garbage collection. It should be mentioned that the solve step of XSB writes the computed answer to a file and that this is included in the recorded times.

3.4 Presenting the Results

3.4.1 Scalable Ambient Programs

When presenting the results obtained for the scalable ambient programs we take the approach of [1]. Thus, we present the results in graphs that show the solve step running time as a function of the size of input program. In order to prevent garbage collection from dominating the benchmarks we present the times that do not include the contributions of the respective garbage collectors.

For the Succinct Solver we know that the execution time is bounded by a polynomial algorithm. For XSB we know that it evaluates ground queries to programs given in Datalog extended with negation with polynomial data complexity [14].

Therefore, as in [1], we assume that measured execution times, t, are explained by a model

$$t = c_1 \cdot p^x + c_0$$

where $x \in \mathbb{R}$, and the factor $p \in \mathbb{N}$ is indicative for the size of the program. We estimate c_0, c_1 and x by a least-square fit of the model to the measured times. Due to separation of initialization and solving we assume c_0 to be 0 since, when $p = 0$ no time is used. We refer to [1] for more details.

3.4.2 Carmel Programs

We treat the results obtained from the Carmel programs differently. The number of different Carmel programs is quite low and it is natural to present the results in tables rather than graphs.

4 Discretionary Ambients: Scalable Router Programs

4.1 The Language

The implementation language of the scalable test programs is that of Discretionary Ambients, which extends mobile ambients with *co-capabilities* and a notion of *types* in the form of *groups*. We will briefly explain the syntax of these new constructs but refer to [1] and [11] for detailed treatments.

As shown in figure 3 the language deals with processes, P, and capabilities, M. We let n range over names and μ range over group names. The construct $(\nu\mu)P$ introduces

$$P \quad ::= \quad (\nu\mu)P \quad | \quad (\nu n : \mu)P \quad | \quad \mathbf{0} \quad | \quad P_1|P_2 \quad | \quad !P \quad | \quad n[P] \quad | \quad M.P$$

$$M \quad ::= \quad \mathtt{in}\ n \quad | \quad \mathtt{out}\ n \quad | \quad \mathtt{open}\ n \quad | \quad \overline{\mathtt{in}}_\mu\ n \quad | \quad \overline{\mathtt{out}}_\mu\ n \quad | \quad \overline{\mathtt{open}}_\mu\ n$$

Figure 3: Syntax of Discretionary Ambients

a new group μ and its scope P and $(\nu n : \mu)P$ introduces a new name n belonging to group μ with scope P. Furthermore, the set of capabilities $\{\mathsf{open}\ n,\ \mathsf{in}\ n,\ \mathsf{out}\ n\}$ is supplemented by a corresponding set of co-capabilities $\{\overline{\mathsf{open}}_\mu\ n,\ \overline{\mathsf{in}}_\mu\ n,\ \overline{\mathsf{out}}_\mu\ n\}$ that names both the granter, n, and the grantee group, μ, of a given capability.

4.2 The Analyses

For the suite of scalable Discretionary Ambient programs we planned to apply both the 0-CFA analysis and the 1-CFA analysis described in [1]. Unfortunately, however, an unexpected problem with XSB Prolog (below) has prevented this.

4.2.1 The 0-CFA

The 0-CFA approximates the behavior of a process by a single abstract configuration describing all possible derivations of the process. It amounts to systematically performing the following approximations:

- The analysis distinguishes between the various groups of ambients but not between the individual ambients.
- The analysis does not keep track of the exact order of the capabilities inside an ambient nor of their multiplicities.
- The analysis represents the tree structure of the processes by a binary relation $\mathcal{I}$ modeling the father-son relationship.

Formally, we define the binary relation $\mathcal{I}$ as a mapping

$$\mathcal{I} : \mathbf{Group} \to \mathcal{P}(\mathbf{Group} \cup \mathbf{Cap})$$

where **Group** is the set of groups, and **Cap** is the set of group capabilities and co-capabilities. The judgments of the analysis have the form

$$\mathcal{I} \models^*_\Gamma P$$

and express that $\mathcal{I}$ is a safe approximation of the configurations that P may evolve into when ambient names are mapped to groups as specified by Γ and when $*$ is the ambient group of the ambient enclosing the process P.

4.2.2 The 1-CFA

The 1-CFA follows the same scheme but adds a context by additionally recording information of grand fathers. Thus the analysis represents the tree structure as a *ternary* relation $\mathcal{I}$ expressing a grand father-father-(grand) son relationship. This is expressed in the ternary relation $\mathcal{I}$ defining the mapping

$$\mathcal{I} : \mathbf{Group} \to \mathcal{P}(\mathbf{Group} \to \mathcal{P}(\mathbf{Group} \cup \mathbf{Cap}))$$

and the judgments of the analysis now become

$$\mathcal{I} \models_{\Gamma}^{*,\top} P$$

As before $*$ denotes the group of the ambient enclosing P while, additionally, $\top$ denotes the group of the ambient enclosing $*$.

Unfortunately the 1-CFA analysis cannot be run in XSB v2.6. It appears that something in the concrete formulation of the derived constraints prevents XSB from distinguishing between fully instantiated and partial answers. The rather unsatisfac-

```
Answer Return ERROR: Failed to unify answer
ret(s_1_1,r,overlineopen(p,r))
with Answer Template:
ret(\"s_1_1\",\"r\",overlineopen(\"s_1_1\",\"r\"))
(* Note: this template may be partially instantiated *)
++Error[XSB]: [Runtime/C] Unequal Constants
! Aborting...
Removing incomplete tables...
```

Figure 4: XSB error message resulting from running the 1-CFA

tory result is the error message shown in figure 4. The problem was acknowledged for v.2.5 of XSB by Luis Fernando P. de Castro, who is a member of the XSB team, and is still on the bug-list for v2.6 of the system.

4.3 Representation of Constraints

The analyses of this section are split into program representation functions and closure conditions. When conducting an analysis of a program the representation function is used to derive a predicate PRG expressing the initial structure of the program in an abstract sense. The closure condition then describes how the relation $\mathcal{I}$ can be computed once PRG is known.

As in [1] the measured solve time only includes the computation of the closure condition. Processing PRG is part of preprocessing.

This technique allows us to easily experiment with the order of conjuncts in preconditions of the analyses. Consider the clause for checking capabilities $\mathsf{in}\ \mu$ in closure condition da0_2:

$$\forall *, \mu, t_1 : \left[\begin{array}{l} \mathsf{in}(\mu) = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array}\right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*, t_1) \wedge \\ \\ \forall \mu^a, \mu^p, t_2 : \left[\begin{array}{c} \overline{\mathsf{in}}(\mu^a, \mu) = t_2 \wedge \\ \mathcal{I}(\mu, t_2) \wedge \\ \mathcal{I}(\mu^p, \mu) \wedge \\ \mathcal{I}(\mu^a, t_1) \wedge \\ \mathcal{I}(\mu^p, \mu^a) \end{array}\right] \Rightarrow \mathcal{I}(\mu, \mu^a) \end{array}\right]$$

we obtain the different analyses used in the following by modifying the inner precondition:

$$\forall \mu^a, \mu^p, t_2 : \left[\begin{array}{c} \overline{\mathsf{in}}(\mu^a, \mu) = t_2 \wedge \\ \mathcal{I}(\mu, t_2) \wedge \\ \mathcal{I}(\mu^p, \mu) \wedge \\ \mathcal{I}(\mu^a, t_1) \wedge \\ \mathcal{I}(\mu^p, \mu^a) \end{array}\right]$$

in the clauses corresponding to in, out, and open. Thus, the analyses used for this experiment are:

da0_1 first recognize a capability, then the path to the root of the redex, and finally the matching co-capability.

da0_2 first recognize a co-capability, then the path to the root of the redex, and finally the matching capability (see appendix E.1).

da0_3 first recognize a co-capability, then the matching co-capability, and finally the path to the root of the redex.

da0_7 As da0_2, but explicit checks for term equality are postponed until the corresponding term variables have been bound (see appendix E.2).

4.4 Test Programs

The programs themselves are the same as those used in [1] and will be described briefly in the following:

1 - a 'packet' ambient travels along a line of m linked 'router' ambients.

s - a 'packet' ambient travels from the upper left corner to the bottom right corner of an $m \times m$ grid of 'router' ambients. Each location (router), which is not bottom- or right-justified, offers advance along two paths - rightwards or downwards. A schematic drawing of such a grid, and the Discretionary Ambient code of an instance $m = 2$ are included in appendices B.1 and B.2.

lvg - a 'packet' ambient travels from the upper left corner to the bottom right corner of an $m \times m$ grid of 'router' ambients. Each location (router) offers advance to the immediate right or to any location on the line below.

sph - a collection of m 'spherical' ambients coexist. Each sphere can enter, exit, or open any other and allows all others to enter, exit, or open itself.

Table 4.4 shows how the size of the universe $|\mathcal{U}|$, the size of the abstract initial configuration (the input program) $|\mathsf{PRG}|$, and the size of the computed solution $|\rho(\mathcal{I})|$ depend on the parameter m. This information allows us to estimate κ for each of the program types.

	$\lvert\mathcal{U}\rvert$	$\lvert\mathsf{PRG}\rvert$	$\lvert\rho(\mathcal{I})\rvert$	κ
$\mathsf{1}(m)$	$6m+7$	$6m+2$	$9m+3$	0
$\mathsf{s}(m)$	$6m^2+7$	$6m^2+2$	$9m^2+3$	0
$\mathsf{lvg}(m)$	$6m^2+5m+7$	$2m^3+5m^2+5$	$2m^3+8m^2+2m+6$	2
$\mathsf{sph}(m)$	$3m^2+5m+1$	$6m^2+m$	$3m^3+4m^2+m$	2

Table 3: Computed characteristics of the various program types.

We expect the Succinct Solver to do well for the classes $\mathsf{1}(m)$ and $\mathsf{s}(m)$ as they induce lightly populated analyses. We don't have similar expectations with respect to $\mathsf{lvg}(m)$ and $\mathsf{sph}(m)$ as they induce heavily populated analyses.

4.5 Test Results

We have repeated the 0-CFA version of the precondition conjunct re-ordering experiment of [1] for each program class (1, s, lvg, sph). By doing this we gather information regarding a number of issues:

- Is one solver clearly superior to the other?
- If not, how does the relative performance depend on analysis population?
- Are both solvers sensitive with respect to clause orderings?
- If so, do the different orderings affect the solvers in the same way?

In this section and the corresponding appendices we present the results in graphs. Unless otherwise stated these graphs are logarithmic plots that show the running time of the various solvers in the various experiments as a function of $|\mathsf{PRG}|$. All of the graphs follow the same naming conventions. The abbreviation SS denotes the Succinct Solver. The abbreviations SA, SH, and XSB denote the Succinct Solver with input on ALFP form, the Succinct Solver with input on Horn expanded form, and XSB Prolog with input on Horn expanded form respectively.

The tests documented in the following have been conducted on a multi-user machine with the following specifications:

Vendor	Sun Microsystems
No. of Processors	2
Processor type	UltraSparc
Processor speed	100MHz
Memory	2GB
Disk IO system	RAID

4.5.1 The $1(m)$ Test Suite

The linear router structure 1 is an example of simple programs that induce lightly populated analyses. As we see from the corresponding plots in appendix A.1 the Succinct Solver clearly performs differently, albeit only by a constant factor, for the various orderings and is pretty sure that closure condition da0_2 is optimal. XSB similarly performs differently for the various orderings but shows almost no difference between da0_2 and da0_7.

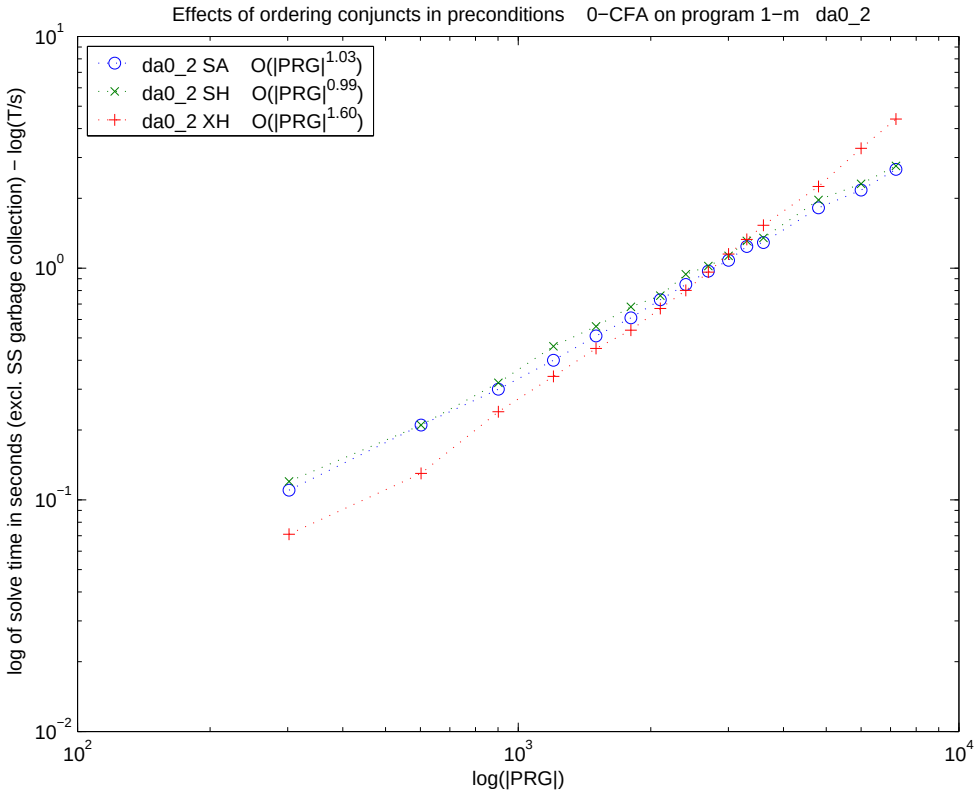


Figure 5: Solve times for Succinct Solver v2.0 ALFP, Succinct Solver v2.0 Horn, XSB v2.6 Horn

As figure 5 shows XSB exhibits the lowest execution times for small program instances. The Succinct Solver, however, shows a much better complexity and as the program instances grow larger we see how, for da0_2, the Succinct Solver actually overtakes XSB.

Also, appendix A.2 shows that the Succinct Solver solves the constraint set in Horn form marginally slower than in ALFP form. Generally, however, they are about equally fast (see figure 5).

4.5.2 The s(m) Test Suite

The simple quadratic router structure described by s also gives rise to lightly populated analyses. As we see from the corresponding plots in appendix B.3 the Succinct

Solver now performs differently almost up to a degree of the complexity polynomial for the various orderings. Closure condition da0_2 is still best. XSB also performs differently for the various orderings but, again, shows almost no difference between da0_2 and da0_7.

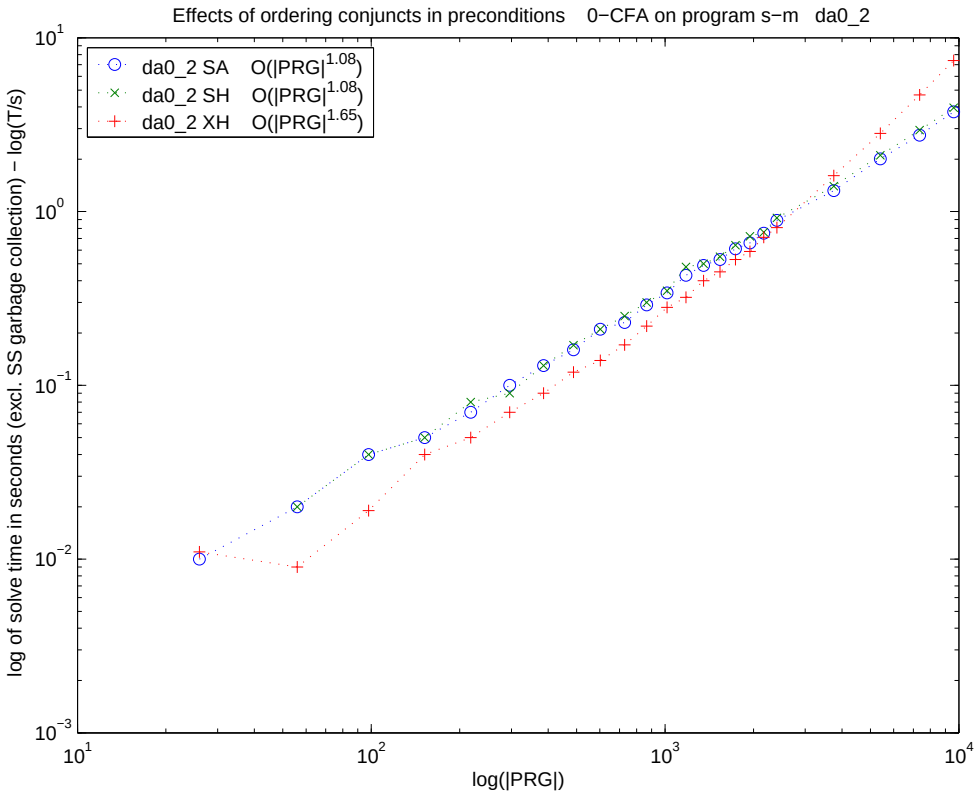


Figure 6: Solve times for Succinct Solver v2.0 ALFP, Succinct Solver v2.0 Horn, XSB v2.6 Horn

The pattern of a lightly populated analysis repeats itself and figure 6 shows that XSB exhibits the lowest execution times for small program instances. As before the Succinct Solver shows a much better complexity and for da0_2 the Succinct Solver actually overtakes XSB for the larger programs.

Appendix B.4 shows that the Succinct Solver performance difference for solving the constraint set in Horn form and ALFP still exists, but is smaller than for 1.

4.5.3 The lvg(m) Test Suite

The complex quadratic router structure described by lvg gives rise to heavily populated analyses. As we see from the corresponding plots for the Succinct Solver in appendix C.1 the performance difference for the various orderings has now shrunk to a constant factor again. Closure condition da0_7 is best, but both da0_3 and da0_2 are almost as good.

XSB exhibits the same behavior, but now it shows almost no difference between da0_2, da0_7, and da0_3.

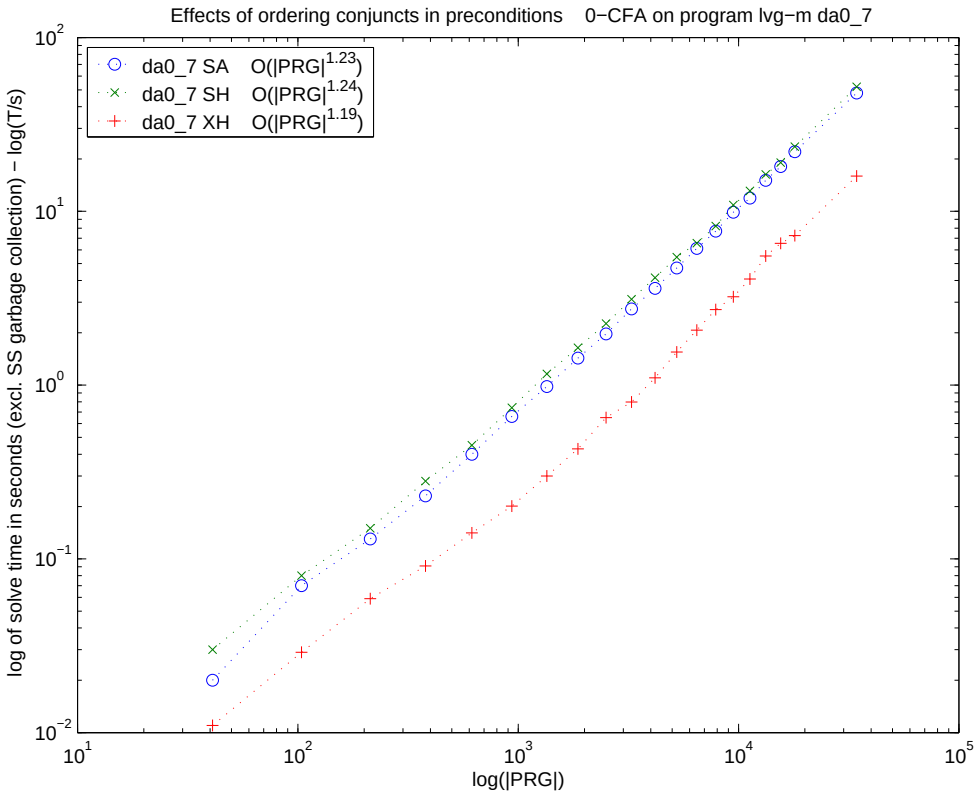


Figure 7: Solve times for Succinct Solver v2.0 ALFP, Succinct Solver v2.0 Horn, XSB v2.6 Horn

XSB now persistently exhibits the lowest actual execution times. As figure 7 shows, however, the complexities of the two solver algorithms now seem to be equal. While the graph clearly indicates that XSB is a constant factor (see also appendix C.2) faster, the weak indication on the complexity polynomial is too small to be significant. The second figure of appendix C.2, which shows the measurements for the Succinct Solver point-wise divided by the measurements for XSB, indicates that XSB has an advantage of about a factor 2.

Figure 7 and the figures of appendix C.2 now indicate that the Succinct Solver performance difference between Horn and ALFP clauses is practically nullified.

4.5.4 The sph(m) Test Suite

The spherical structure described by sph gives rise to analyses that are even more heavily populated.

The plots of appendix D.1 show that neither the Succinct Solver nor XSB show any real performance difference for the various orderings anymore. For the first time, however, the Succinct Solver shows the best (but by an insignificant margin) complexity for da0_1 while XSB still champions da0_7.

Again, as figure 8 shows, the complexities of the two solver algorithms is reasonably equal but XSB (also appendix D.2) has a constant factor in its favor. The small

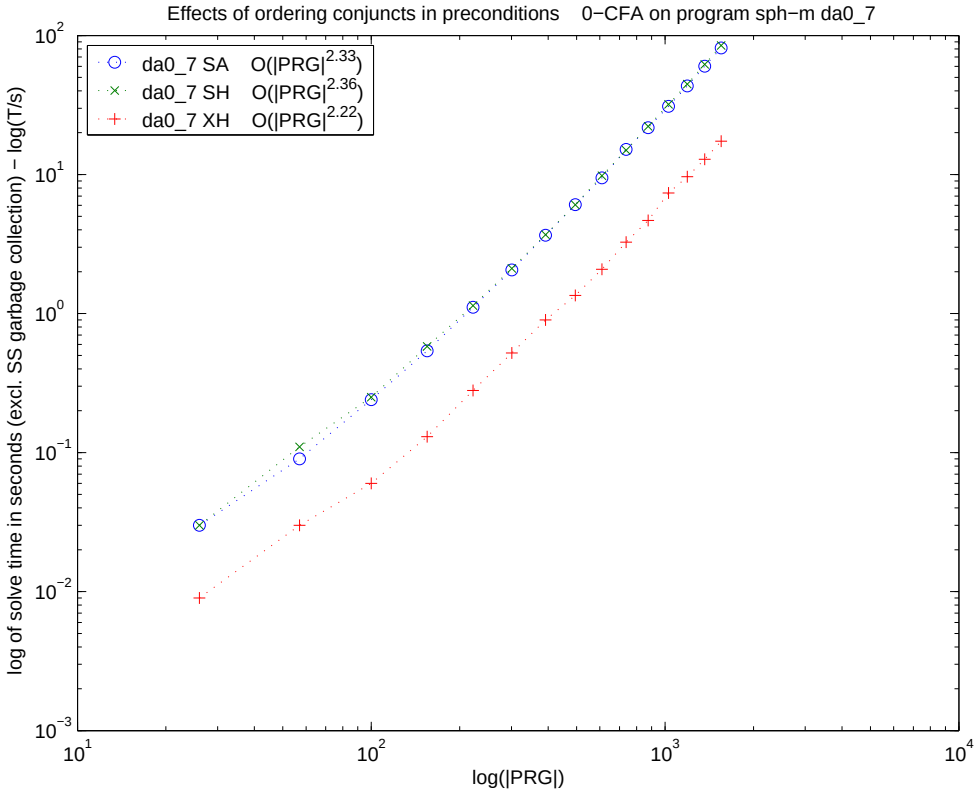


Figure 8: Solve times for Succinct Solver v2.0 ALFP, Succinct Solver v2.0 Horn, XSB v2.6 Horn

indication on the complexity polynomial is still too small to be significant. The second figure of appendix D.2, which shows the measurements for the Succinct Solver point-wise divided by the measurements for XSB, indicates that the constant factor advantage of XSB does not exceed 5 for the program sizes included in the test.

As indicated by figure 8 and the figures of appendix D.2 Horn expanded and ALFP forms are now equal.

4.6 Discussion

The test results of this section reveal quite a bit of useful information about the solvers.

It is evident that the two solvers are not equally good for all types of analyses. The Succinct Solver clearly exhibits the lower asymptotic complexity for analyses that are lightly populated. For heavily populated analyses XSB performs best. In these cases XSB proves to be a small constant factor better but fails to show any significant indication on the complexity polynomials. These results are well in line with the design criteria of the Succinct Solver but it is, however, remarkable that the functional implementation of the Succinct Solver algorithm outperforms XSB. This underlines the importance of good algorithm design and proves that the design choices have had the desired effect.

Notably, XSB still completes most of the tests faster than the Succinct Solver. There is a likely explanation for this. The SLG-WAM abstract machine at the core of XSB is a heavily optimized low-level execution engine implemented in C for which XSB programs are specifically compiled and optimized. The Succinct Solver on the other hand is implemented in SML and, thus, has at least one more interpretation layer to deal with when executing. So, while XSB has an advantage with respect to pure evaluation speed the results show that the design strategy of the Succinct Solver has been successful.

We see that the two solvers basically agree on the best closure condition in all cases. From this we conclude that the overall evaluation strategies (L-to-R) of the two solvers are somewhat alike. That XSB differentiates less between closure conditions da0_2 and da0_7 than the Succinct Solver may be due to XSB's flexible run-time rule selection.

Overall, the results imply that optimization strategies based on the syntactical rewriting/restructuring of clauses, e.g. as suggested in [1], that work well for the Succinct Solver are likely to work for XSB also.

Finally, the results demonstrate that for the Succinct Solver ALFP Logic actually lends to a more efficient evaluation w.r.t. the investigated analyses (albeit only by a small margin) than does Horn logic. This could be explained by the fact that the Horn expansion actually widens the scope of some quantifiers, i.e. in some sense ALFP is more precise.

5 Carmel: Demoney - The Electronic Java Card Purse

5.1 The Language

The Carmel language does not need much of an introduction. It is the Java Card Virtual Machine Language (JCVML) variant considered by the SecSafe project. It constitutes a rational reconstruction of the JCVML used in actual Java Smart Cards but has substantially less instructions.

We will not go into further detail here, but refer to [6] for the syntax and [16] for the semantics of Carmel.

5.2 The Analysis

The analysis applied to Carmel programs is the *Control Flow Analysis* defined in [4] and [3]. The analysis comprise judgments of the form

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models \texttt{addr} : \texttt{instr}$$

where $\hat{S} \in \widehat{\mathsf{Stack}}$, $\hat{L} \in \widehat{\mathsf{LocHeap}}$, $\hat{H} \in \widehat{\mathsf{Heap}}$, $\hat{K} \in \widehat{\mathsf{StaHeap}}$, $\texttt{addr} \in \mathsf{Addr}$, and `instr` is the instruction at `addr`. The meaning of these judgments is that $(\hat{K}, \hat{H}, \hat{L}, \hat{S})$ is an *acceptable* analysis for the instruction `instr` at address `addr`.

The abstract static heap, $\widehat{\mathsf{StaHeap}}$, and abstract heap, $\widehat{\mathsf{Heap}}$, estimates computed by the analysis, which together constitute the abstract global heap estimate, are not likely to become densely populated . The stack, $\widehat{\mathsf{Stack}}$, and the local heaps, $\widehat{\mathsf{LocHeap}}$, however, are as the model associates a separate estimate of each with every instruction of a program.

The domains treated by the analysis are abstract versions of the concrete domains used in the semantics. Some of the abstractions are quite coarse; e.g. object references are simply abstracted into their class and array references are abstracted into their element-type. Semantic information not important to the analysis is simply ignored. This increases the legibility of the analysis, but also prevents it from becoming prohibitively complex.

This analysis and its implementation in the Karma constraint generator is intended to serve as a feasibility study and a benchmarking vehicle to aide in the evaluation of the program analysis approach to Java Card security. In this report they will serve as exactly that and also as a means to compare the Succinct Solver and XSB.

5.3 Constraints and Translation

In order to generate the constraints for the investigated Carmel programs we use the Karma v.2.0pre3-alpha Carmel constraint generator for the Succinct Solver. This implementation is vaguely similar to that described in [5], but the work has made significant progress since then.

5.3.1 Computing with Negative Information

The generator now uses a special relation IMPL to keep track of the locally defined functions of a class. This information is then used negatively in order to compute the *inherited* functions exposed by the class. If an inherited function shares its name

with a locally defined function then the local definition supersedes the inherited one. Obviously, the use of IMPL is subject to the usual stratification requirements, but this is neither a problem for the Succinct Solver nor for XSB.

5.3.2 Computing with a Bounded Universe

The generator was originally devised for the Succinct Solver v.1.0, which was not able to expand the term universe dynamically. Since terms are used as stack indices and the stack must be expected to expand dynamically throughout the course of analysis this is a potential problem.

Luckily Carmel programs that potentially expand the stack infinitely are illegal and prohibited by the byte-code verifier. Also, given a program the verifier is able to compute the maximum depth that the stack may reach during the course of evaluation. We can then use this estimate to prepare a suitable set (sub-universe) of index terms.

The initial translation from ALFP to Normal programs did not translate explicit explicit tests for equality outlined in section 2.3.2. Thus, XSB could freely generate new stack indices whenever necessary, which is theoretically fine as only a finite number is required. We did, however, encounter termination problems as XSB was not able to finish when analyzing a number of the test programs. Therefore the translation was augmented as outlined in section 2.3.2. The source of the unexpected behavior will be examined in section 5.5.

5.4 Test Programs

The *Demoney* application by Trusted Logic [8][7] is a demonstrative electronic purse for Java Smart Cards meant to be used as a case study in the SecSafe Project. While this program is not a fully grown commercial electronic purse it is sufficiently close and exhibits both the size and the security characteristics relevant to such applets.

The programs that we analyze in this section are all related to the Carmel incarnation of the DEMONEY-STAND-ALONE applet as described in [8]. In the following the programs are treated in order of their sizes - smallest first:

demoney is the demoney stand-alone applet with all necessary API functionality given as *native* function declarations. References to local object variables are fully qualified.

demoney-local is similar to demoney except references to local object variables are *not* fully qualified.

jc212api is a partial implementation of the Java Card API v.2.12 supplied by Trusted Logic. A number of functions are only present as *native* declarations and the openplatform/globalplatform classes are not implemented.

API is the jc212api with simplistic (dummy) implementations of all functions previously given as *native* declarations and the openplatform/globalplatform classes.

demoney-impl-api is similar to demoney but includes the full API implementation. This is the full version of the program and is the primary object of study in this section.

5.4.1 The Implementation of *native* Functions

As mentioned some of the programs are not fully implemented. A number of functions are simply declared as *native* functions and, thus, their implementations are presumed to be given in another language and exposed through the Java Card Runtime Environment, JCRE. The Carmel implementations do not distinguish between normal and *native* function definitions, but for the analysis to be complete in some sense we need to supply code for the *entire* system.

Therefore, we give a dummy Carmel definition of all *native* functions. The ideal solution would be to implement a 'correct' emulation of the JCRE functionality in question. However, based on the assumption that none of the *native* functions invoke call-back functions in the actual program (thus creating an unexpected flow of control), we have chosen to give each function a minimal implementation, that influences the analysis as little as possible.

The *native* functions have been implemented according to a simple methodology that takes nothing but the declared types of the results of the functions into account.

Target Type	Implementation
void	`0: return`
boolean	`0: push s 1` `1: return s`
short	`0: push s 0` `1: return s`
byte	`0: push s 0` `1: return s`

τ[]	`0: push s 10`
	`1: new` τ`[]`
	`2: return r`

class/interface	`0: push r null` `1: return r`

Table 4: The Correspondence between Types and Implementations of *native* functions.

Table 4 shows the correspondence between target/result types and implementations of *native* functions. In the table τ[] ranges over all array reference types and class/interface ranges over all class/interface reference types.

5.5 Test Results

We have conducted a number of tests for each of the described test-programs.

As mentioned in section 5.2 the control-flow analysis tracks four distinct relations. We also mentioned in section 3.1.2 that the order in which we query XSB for the solutions of these relations is likely to influence the solver performance. In order to test this hypothesis we have bench-marked XSB for each possible query order on all of the programs. The results of these tests are placed in appendices I-J.

In order to compare the performance of the Succinct Solver and XSB we have bench-marked the Succinct Solver for ALFP- and Horn-form on all of the programs. The comparison between the Succinct Solver benchmarks and the best obtained XSB benchmark for each program will be presented in the following sections.

Finally, in order to track the source of the observed non-termination problem we have tested the Succinct Solver for a number of stack-bounds between five and thirty-six on all of the programs. The results of these tests are also placed in appendices I-J.

All of the results of this section and the corresponding appendices are presented in tables. These tables all have common naming conventions. The abbreviated solver names SS, SH, and XSB denote the Succinct Solver with input in ALFP form, the Succinct Solver with input in Horn expanded form, and XSB with input in Horn expanded form. The entries $|\hat{S}|$, $|\hat{H}|$, $|\hat{L}|$, and $|\hat{K}|$ denote the sizes of the stack, heap, local heap, and static heap estimates respectively. The entries CPU, Init, and Time denote the CPU-time spent by the solve step, the initialization time including garbage collection, and the total time for the entire process including garbage collection respectively. The entry Order denotes the order in which the relations were queried for XSB to obtain the corresponding computation time.

The tests documented in the following have been conducted on a single-user machine with the following specifications:

Vendor	DELL
No. of Processors	1
Processor type	Intel Mobile Pentium 4
Processor Speed	2.0 GHz
Memory	512MB
Disk IO system	IDE 5400RPM

5.5.1 Demoney - no API Implementation

As previously mentioned we have used two versions of the demoney program with the API given as native functions. The only difference between the two programs lie in the degree of qualification of references to variables and methods local to the current object (this).

We expect these differences to be eliminated by the constraint generator and, thus, the analyses of the two programs to yield identical results in identical times. This is indeed the case. The generated constraints are the same. Table 5 below and table 11 in appendix G show that the size of the solutions are identical and that computation times are practically identical.

Solver	$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
SS	1379	14	3016	0	2.11	4.30	9.7 0	
SH	1379	14	3016	0	2.11	4.59	8.77	
XSB	1379	14	3016	0	1.78	8.27	10.41	l-k-s-h-

where sol(SS) $\subseteq$ sol(XSB) and sol(XSB) $\subseteq$ sol(SS).

Table 5: Results for the demoney program.

Table 5 shows that the Succinct Solver and XSB completely agree on the result and that XSB is marginally faster than the Succinct Solver. The two assertions (sol(..) $\subseteq$ ..) below the table denote the result of a two-way inclusion test of the solution for the Succinct Solver, denoted sol(SS), and the solution for XSB, denoted sol(XSB). We also see that the relationship between CPU time expenditures of

the solving step is not necessarily indicative for the relationship between total time expenditures of the whole process. In fact it seems that the Succinct Solver spends a majority (about $\frac{2}{3}$) of its time garbage collecting while XSB spends it initializing.

Tables 9 and 11 in appendices F and G show that XSB performs best for this problem if the relations are queried for in the order $\hat{L} - \hat{K} - \hat{S} - \hat{H}$. As evident from table 11 the best query order does not necessarily reveal much information. Simply observing the impact of the first ordering component on the ordering of the results in e.g. table 12 may be more informative.

Finally, table 10 in appendix F indicates that the there is no non-termination problem for these programs. As could be expected we observe that more permissive stack bounds give lower performance due to the larger universe/Herbrand Base.

5.5.2 The API Implementations

We also have two API implementations that are, however, genuinely different. The jc212api is the simpler of the two implementations, which is reflected in tables 6 and 7.

Solver	$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
SS	1845	33	4662	37	2.46	4.48	11.41	
SH	1845	33	4662	37	2.35	4.46	10.58	
XSB	1845	33	4662	37	2.40	8.17	10.93	k-l-s-h-

where sol(SS) $\subseteq$ sol(XSB) and sol(XSB) $\subseteq$ sol(SS).

Table 6: Results for the jc212api program.

For these programs the Succinct Solver is marginally faster than XSB; the difference, however, seems to be negligible. As expected the Succinct Solver performs about the same for clauses in ALFP and in Horn format. For both of the programs the solvers agree on the solution. Again, we notice that the total times spent by the two solvers are largely equal while the Succinct Solver spends it garbage collecting and XSB spends it initializing - the CPU times remain bad predictors of the final outcome.

Solver	$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
SS	2251	40	5404	39	2.65	4.77	12.2 0	
SH	2251	40	5404	39	2.71	4.81	11.37	
XSB	2251	40	5404	39	2.75	8.31	11.44	k-h-s-l-

where sol(SS) $\subseteq$ sol(XSB) and sol(XSB) $\subseteq$ sol(SS).

Table 7: Results for the API program.

As the tables show XSB performs best in both cases if $\hat{K}$ is queried first. For the two programs the best orders seem to disagree on the remaining components. If we interpret tables 14 and 17 of appendices H and I, however, $\hat{K} - \hat{H} - \hat{S} - \hat{L}$ seems to be a good initial guess..

When the analysis was initially run on these programs rather strange behavior was observed. As we see from tables 15 and 18 in appendices H and I the analysis

results seemed to reflect unlimited stack growth. Subsequent investigation revealed that the growth occurred in two functions that were both overloaded.

```
package com.sun.javacard.impl ...;
public class AppTable {
  static byte findApplet(javacard.framework.AID);
  public static byte findApplet1(byte[], short, byte);
}

package javacard.framework ...;
public final class AID {
  public boolean equals(java.lang.Object);
  public boolean equals1(byte[], short, byte);
}
```

Figure 9: Overloaded functions that caused stack problems before α-renaming.

The explanation of this behavior lies in the specification/implementation of the applied control flow analysis. The analysis is not designed to distinguish overloaded functions. Theoretically, this does not pose a problem as the analysis specification has been proved correct with respect to the Carmel semantics. In practice, however, the implementation appears not to handle overloaded functions particularly well.

In order to overcome the problem we simply α-renamed overloaded functions apart whenever necessary, which completely eliminated the problem as shown in tables 16 and 19.

5.5.3 Demoney - full API Implementation

The full implementation of demoney including the full API implementation is the primary subject of this investigation. As the program is a combination of two of the ones already treated we have some apriori knowledge about what to expect - e.g. that problems with stack bounds are likely to occur.

The program also possesses some characteristics that none of the other programs did. First of all the program is large - large enough to have industrial relevance. Secondly, we have supplied a Main class to the program with a method that is responsible for installation of the demoney applet. We also make sure to invoke this function in order to exercise the program code. This added code is shown in figure 10.

```
package dk.imm.main { 0xa0, 0x24, 0x24, 0x24, 0x24, 0x42 };

public class Main {
 public static void doit(short) {
   0: load r 1
   1: push s 0
   2: push s 42
   3: arraystore b
   4: load r 1
   5: push s 87
   6: push s 42
   7: invokedefinite fr.trustedlogic.demo.demoney.Demoney.install(byte[],short,byte)
} }

#pragma verbatim{
  &
  L(mid(cl_dk_imm_main_Main,mn_doit),pc_0,var_1,ar_BYTE__SPECIAL)
}
```

Figure 10: Main class and invocation clause as added to demoney-impl-api.

As expected table 8 shows that the analysis result is of about twice the size of any of the previous ones. It is also evident that the solvers agree on the solution and that the Succinct Solver computes it about twice as fast as XSB.

However, the overall time expenditure profile of the two solvers does not change and the CPU times remain bad indicators of the total times spent.

Solver	$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
SS	4262	75	11103	39	6.60	10.90	32.52	
SH	4262	75	11103	39	6.36	10.16	28.86	
XSB	4262	75	11103	39	13.37	16.25	30.24	s-h-k-l-

where sol(SS) $\subseteq$ sol(XSB) and sol(XSB) $\subseteq$ sol(SS).

Table 8: Results for the demoney-impl-api program.

The table indicates that the optimal query order has changed substantially. This is backed up by the results shown in table 20 of appendix J, which indicate that $\hat{S} - \hat{H} - \hat{L} - \hat{K}$ is a good initial guess at an optimal query order.

Furthermore, tables 21 and 22 show that α-renaming was necessary in this case also.

5.6 Discussion

Regarding XSB prolog, it is evident that the order in which the analyzed relations are queried is important for performance. It is, however, not similarly clear which order is best; this seems to depend on the program to be analyzed. It may be that the determining factor is the size of the program but from the results we cannot tell. If we, however, assume this and regard the data presented here as indicative then $\hat{L}$ and $\hat{K}$ would be the relations to query first and second for small programs, such as the demoney variants without API code. For slightly larger programs, such as the API implementations, $\hat{K}$ and $\hat{H}$ have the largest impact. And finally, for large programs, such as the full demoney-impl-api implementation, $\hat{S}$ and $\hat{H}$ carry the greatest weight. The best advise, if any, one could give on this background is that large programs should probably be queried in the order $\hat{S} - \hat{H} - \hat{L} - \hat{K}$.

We have already established that for programs that yield lightly populated analyses XSB is the fastest solver for small programs and the Succinct Solver is the best solver for larger programs. The results showed in this section may indicate that the analyzed Carmel programs yield such lightly populated analyses. The fact that XSB performs slightly better for the smaller program, whereas the Succinct Solver is clearly superior (in terms of CPU time) for the large program would then be a reasonable reflection of this. As mentioned in section 5.5.3 the Succinct Solver spends a substantial amount of its work on garbage collection, which is somewhat concerning. As the physical memory of the system is never exhausted we attribute this to the garbage collection policy of New Jersey SML.

The implications of the results are two-fold. Firstly, we have established that the use of program analysis on Carmel programs of industrial proportions is indeed feasible as both solvers complete the analysis for even the largest program in a very short amount of time. Secondly, the results are quite favorable for the Succinct Solver as it finishes about twice as fast as XSB (CPU-time) for the largest program. While this one example does not provide the substance to actually support the design assumption of the Succinct Solver - i.e. that it should perform well for benign problems as they often arise in static analysis - it does not oppose it either. In short the Succinct Solver has performed well in the experiments of this section and it seems to constitute a very reasonable technology choice for the SecSafe project.

Finally, as a small side-note regarding the analysis, we observe that the results of the stack size experiment show that computing with predetermined upper bound on the size of the stack was not a bad idea. It was really helpful in pinpointing a potentially problematic peculiarity of the analysis implementation that we could then easily fix by hand. Had we allowed unbounded expansion of the stack we would have experienced non-termination, which would have been much less helpful.

6 Conclusion

We have submitted the *Succinct Solver v2.0* and *XSB Prolog v2.6* to a number of comparative tests. These tests have been based on a number of scalable *Discretionary Ambient programs* and a number of *Carmel programs*. We have applied a *Horn expansion* procedure in order to translate formulas of *ALFP Logic* to *Normal programs*, thus enabling the reuse of existing analyses and their corresponding ALFP constraint generators. The obtained results show that the Succinct Solver at worst performs a small constant factor worse than XSB Prolog. In optimum cases the Succinct Solver outperforms XSB Prolog by having a substantially lower asymptotic complexity.

On the approach It is evident that there are quite a number of factors that are important for the performance of the compared solvers.

Obviously, the unique combination of settings, input optimizations, etc., which is necessary in order to obtain the absolute peak performance of any of the solvers, is not likely to be obtained by a user that does not have intimate design knowledge of the system in question. In our opinion most users (even expert users) of technology do not have such knowledge and, in the case of XSB Prolog, neither do we.

In order to deal with this we have approached both solvers with a user mind-set and made an effort to test the solvers on even ground. We have used the Horn expansion approach to clause generation, which allows the solvers to be tested on practically identical input. Also, we have avoided the use of solver-specific optimizations - except for tabling, which is crucial to the behavior of XSB Prolog.

We are confident that our results give a good indication of what expert users would be able to obtain from these solvers. However, one should keep in mind that both solvers may have an untapped potential as optimizations are likely to exist.

On the results From a performance perspective it is clear from the obtained results that both XSB and the Succinct Solver are capable solvers. XSB seems to have a more efficient implementation and is superior when analyzing programs that are small or induce heavily populated analyses. In the later case, however, the advantage of XSB is at worst a constant factor. In the most extreme case investigated here this constant factor has been less than five and in the more reasonable case less than two.

The Succinct Solver clearly has the better algorithm for analyzing programs that induce lightly populated analyses. The CPU-time measurements show that for large problems this better algorithm more than makes up for the more efficient low-level implementation of XSB. In the context of the Carmel experiments we observe a slightly disturbing behavior of the New Jersey SML garbage collector, which uses more CPU-time than the solver algorithm itself. In terms of total running time, however, the Succinct Solver still performs at least as well as XSB in these experiments.

From an analysis implementation perspective both solvers have advantages as well as disadvantages. The ALFP syntax of the Succinct Solver constitutes a rich and convenient notation for implementations of flow-logic based analyses. In particular the Succinct Solver notion of finite universes allows for a flexible and convenient use of explicit checks for term equality/inequality in preconditions. It should be noted, however, that the nice ALFP notation comes at the price of a strict and quite rigid stratification requirement.

The syntax of XSB is able to express certain higher order concepts involving functionally parameterized predicates. Also, the the stratification requirements imposed

on programs are quite flexible - depending on the tabling mode. In general, however, the General Logic syntax is less expressive than ALFP Logic and explicit checks for term equality/inequality are awkward to express as the notion of finite universes is not there to help. Finally, the relations relevant to the analysis must be queried manually in a not completely obvious order for the solver to perform well.

A general highlight is that both solvers adhere to a similar left-to-right evaluation strategy, which allows a number of optimization strategies that involve syntactical reordering of clauses to be efficient for both.

With respect to SecSafe and the study of Carmel the results are very positive. Both of the solvers perform well on the studied control flow analysis - we recorded between 30 and 37 seconds of total running time for XSB (depending on query order) while the Succinct Solver finished in less than 29 seconds. We take these results as a clear indication that the static analysis approach to Carmel program verification is feasible and likely to be scalable.

For small programs XSB performs slightly better, but for large programs the Succinct Solver shows the largest potential by a great margin. Given that the applied Carmel analysis is a fully grown realistic control flow analysis this does nothing to oppose the assumption regarding the relative benignity of static analyses that underlies the design of the Succinct Solver. So, while in the context of SecSafe the Succinct Solver has shown the best performance only by a small margin, it has also shown a potential that justifies the effort put into it.

References

[1] Mikael Buchholtz, Hanne Riis Nielson, and Flemming Nielson. Experiments with succinct solvers. Technical Report SECSAFE-IMM-002-1.0, IMM - Technical University of Denmark, February 2002.

[2] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 1989.

[3] René Rydhof Hansen. Extending the flow logic for carmel. Technical Report SECSAFE-IMM-003-1.0, IMM - Technical Univarsity of Denmark, june 2002.

[4] René Rydhof Hansen. Flow logic for carmel. Technical Report SECSAFE-IMM-001-1.5, IMM - Technical Univarsity of Denmark, June 2002.

[5] René Rydhof Hansen. Implementing the flow logic for carmel. Technical Report SECSAFE-IMM-004-1.0, IMM - Technical University of Denmark, November 2002.

[6] Reneaud Marlet. Syntax of the jcvm language to be studied in the secsafe project. Technical Report SECSAFE-TL-005-1.7, Trusted Logic, 2001.

[7] Reneaud Marlet. Demoney: A demonstrative eletronic purse - card specification -. Technical Report SECSAFE-TL-008-0.8, Trusted Logic, November 2002.

[8] Reneaud Marlet. Demoney: Java card implementation. Technical Report SECSAFE-TL-008-0.8, Trusted Logic, November 2002.

[9] F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *10th European Symposium on Programming (ESOP)*, number 2028 in LNCS, pages 252–268. Springer Verlag, 2001.

[10] Flemming Nielson, Hanne Nielson, and Helmut Seidl. A succinct solver for alfp. *Nordic Journal of Computing*, (9):335–372, 2002.

[11] Hanne Riis Nielson and Flemming Nielson. Discretionary ambients. Manuscript, 2001.

[12] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog.* John Wiley, 2 edition, 1995.

[13] Konstantino Saganos, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Boaqiu Cui, and Ernie Johnson. *The XSB System - Version 2.5 - Programmers Manual*, March 2002.

[14] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):586–634, 1998.

[15] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Xsb as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 442–453. ACM Press, 1994.

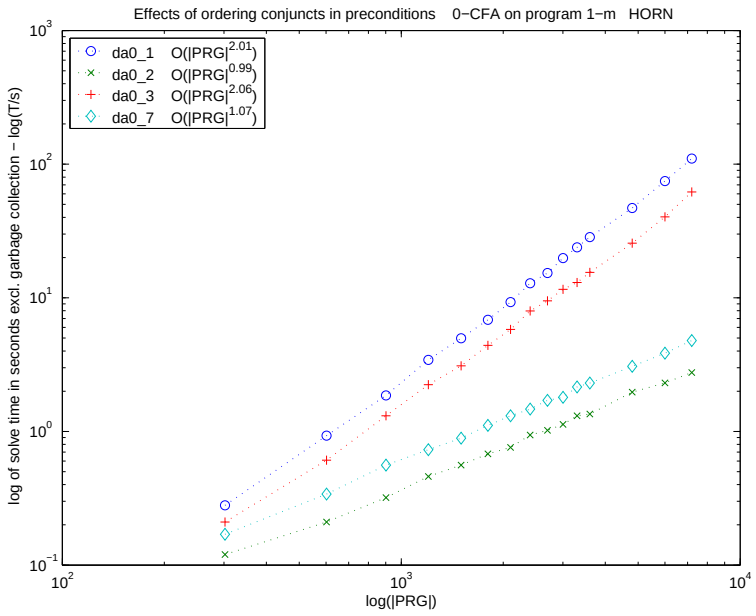
[16] Igor Siveroni and Chris Hankin. A proposal for the jcvmle operational semantics. Technical Report SECSAFE-ICSTM-001-2.2, ICSTM - Imperial College, October 2001.

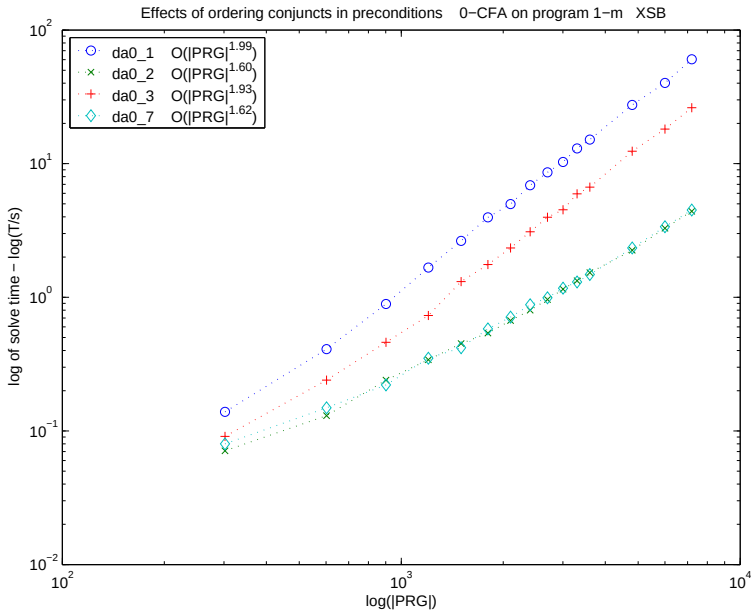
[17] Hongyan Sun. User guide for the succinct solver (v.2.0). Draft, February 2003.

[18] David S. Warren. Programming in tabled prolog (very) draft. Unfinished book on XSB programming, July 1999.

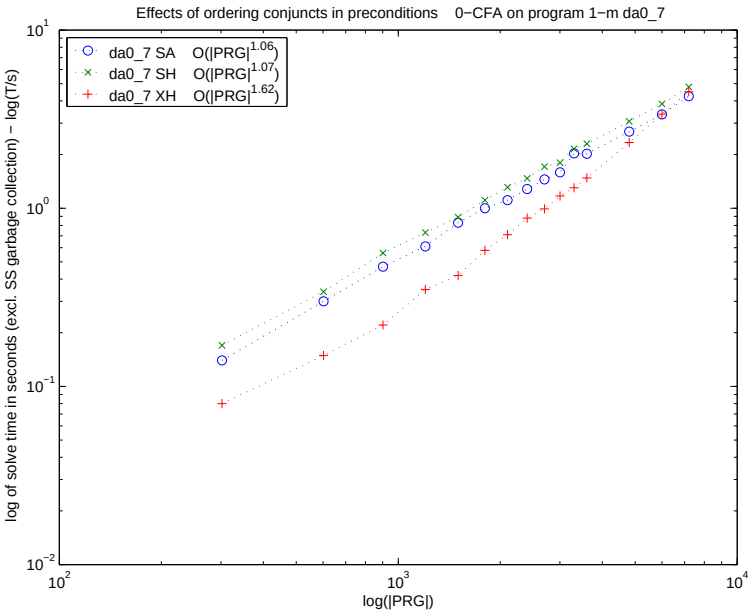
A 0-CFA for the $1(m)$ Programs

A.1 Individual Solvers



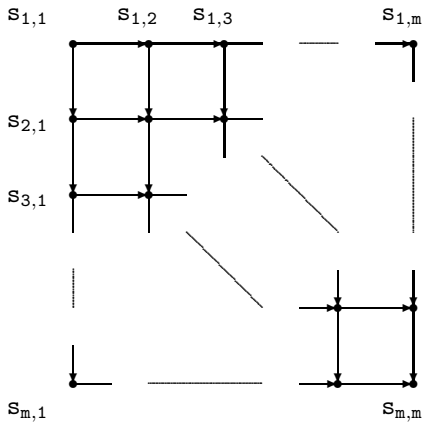


A.2 Solvers Compared



B 0-CFA for the s(m) Programs

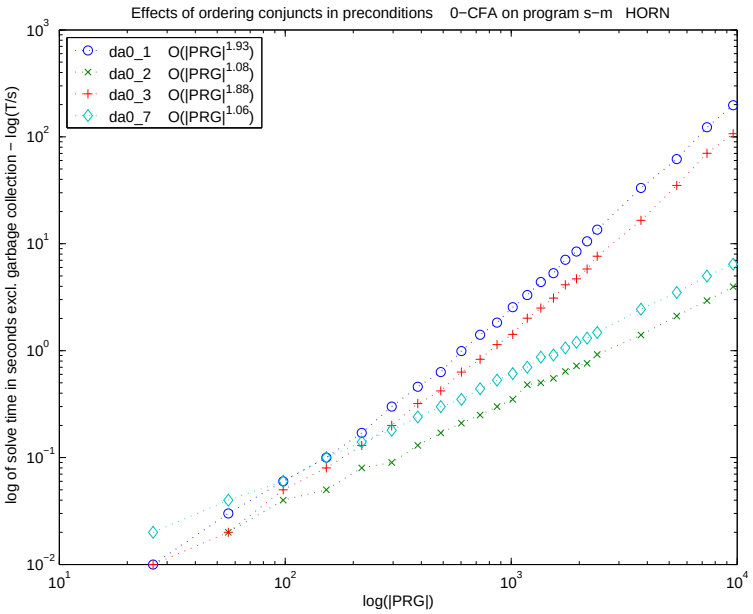
B.1 Schematic Drawing

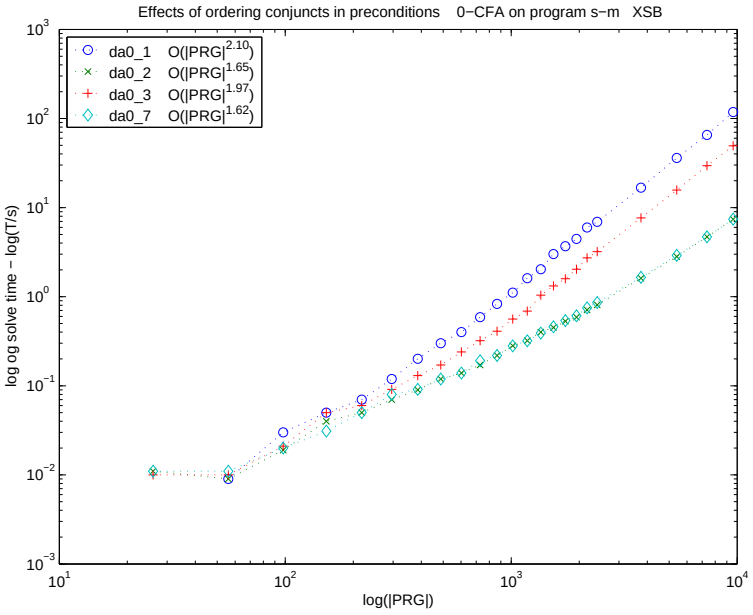


B.2 Problem Instance Code for $m = 2$

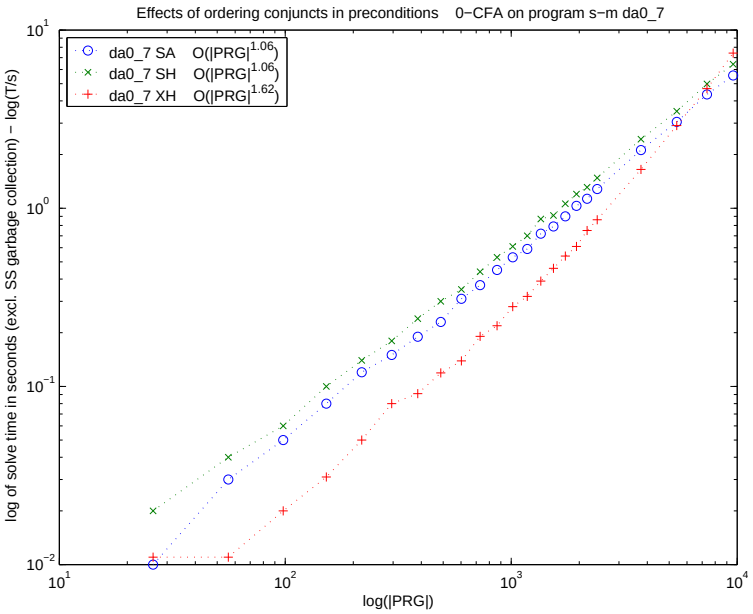
$(\nu \mathtt{p} : \mathbf{P})(\nu \mathtt{r} : \mathbf{R})(\nu \mathtt{s}_{1,1} : \mathbf{S_{1,1}})(\nu \mathtt{s}_{1,2} : \mathbf{S_{1,2}})(\nu \mathtt{s}_{2,1} : \mathbf{S_{2,1}})(\nu \mathtt{s}_{2,2} : \mathbf{S_{2,2}})$
$\quad \mathtt{p}[\mathtt{in}\ \mathtt{s}_{1,1} \mid !\ (\overline{\mathtt{in}}_{\mathbf{R}}\ \mathtt{p}.\ \mathtt{open}\ \mathtt{r})]$
$\mid\ \mathtt{s}_{1,1}[!\ \overline{\mathtt{in}}_{\mathbf{P}}\ \mathtt{s}_{1,1}.\ \overline{\mathtt{out}}_{\mathbf{P}}\ \mathtt{s}_{1,1}$
$\quad \mid\ !\ \mathtt{r}[\mathtt{in}\ \mathtt{p}.\ \overline{\mathtt{open}}_{\mathbf{P}}\ \mathtt{r}.\ \mathtt{out}\ \mathtt{s}_{1,1}.\ \mathtt{in}\ \mathtt{s}_{2,1}]$
$\quad \mid\ !\ \mathtt{r}[\mathtt{in}\ \mathtt{p}.\ \overline{\mathtt{open}}_{\mathbf{P}}\ \mathtt{r}.\ \mathtt{out}\ \mathtt{s}_{1,1}.\ \mathtt{in}\ \mathtt{s}_{1,2}]]$
$\mid\ \mathtt{s}_{1,2}[!\ \overline{\mathtt{in}}_{\mathbf{P}}\ \mathtt{s}_{1,2}.\ \overline{\mathtt{out}}_{\mathbf{P}}\ \mathtt{s}_{1,2}$
$\quad \mid\ !\ \mathtt{r}[\mathtt{in}\ \mathtt{p}.\ \overline{\mathtt{open}}_{\mathbf{P}}\ \mathtt{r}.\ \mathtt{out}\ \mathtt{s}_{1,2}.\ \mathtt{in}\ \mathtt{s}_{2,2}]]$
$\mid\ \mathtt{s}_{2,1}[!\ \overline{\mathtt{in}}_{\mathbf{P}}\ \mathtt{s}_{2,1}.\ \overline{\mathtt{out}}_{\mathbf{P}}\ \mathtt{s}_{2,1}$
$\quad \mid\ !\ \mathtt{r}[\mathtt{in}\ \mathtt{p}.\ \overline{\mathtt{open}}_{\mathbf{P}}\ \mathtt{r}.\ \mathtt{out}\ \mathtt{s}_{2,1}.\ \mathtt{in}\ \mathtt{s}_{2,2}]]$
$\mid\ \mathtt{s}_{2,2}[\overline{\mathtt{in}}_{\mathbf{P}}\ \mathtt{s}_{2,2}]$

B.3 Individual Solvers



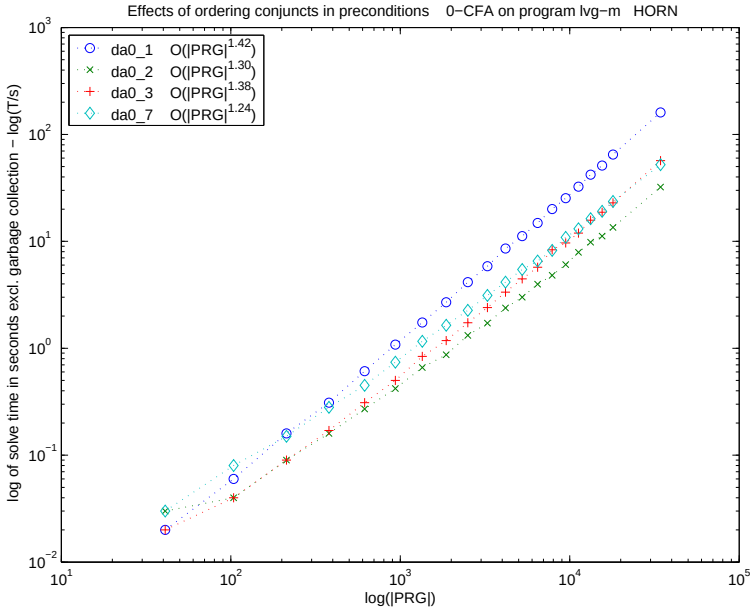


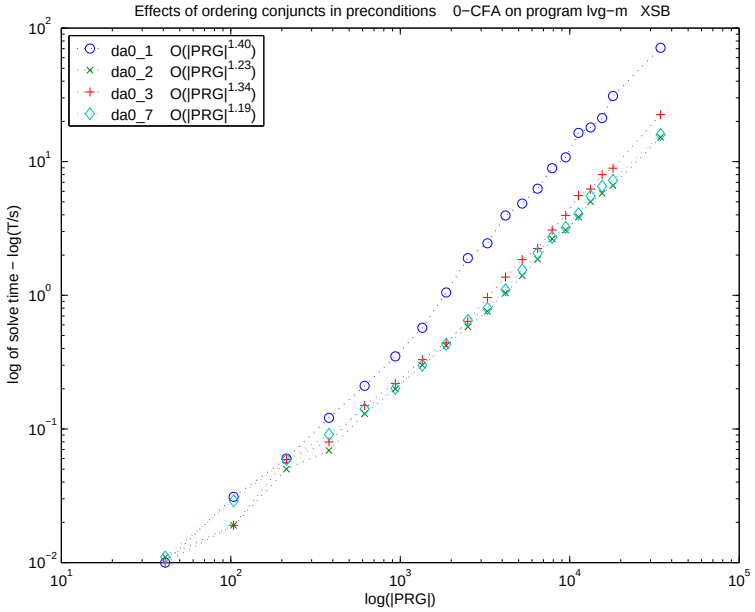
B.4 Solvers Compared



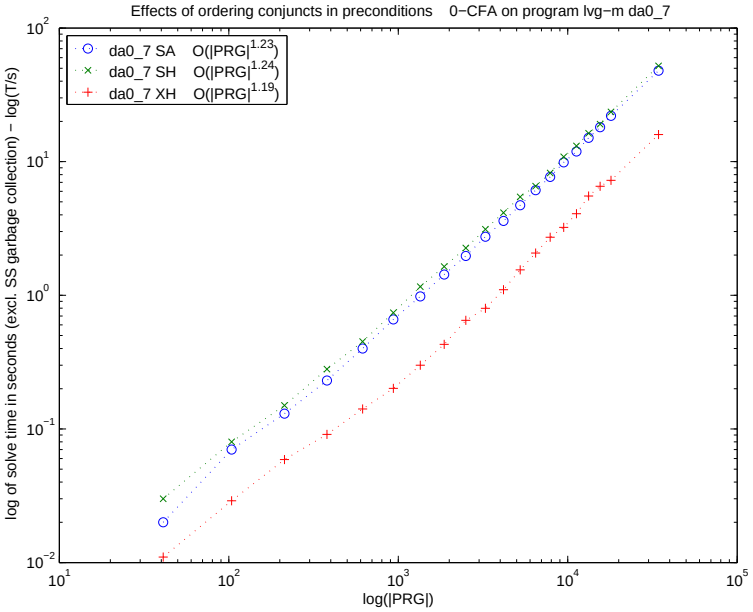
C 0-CFA for the lvg(m) Programs

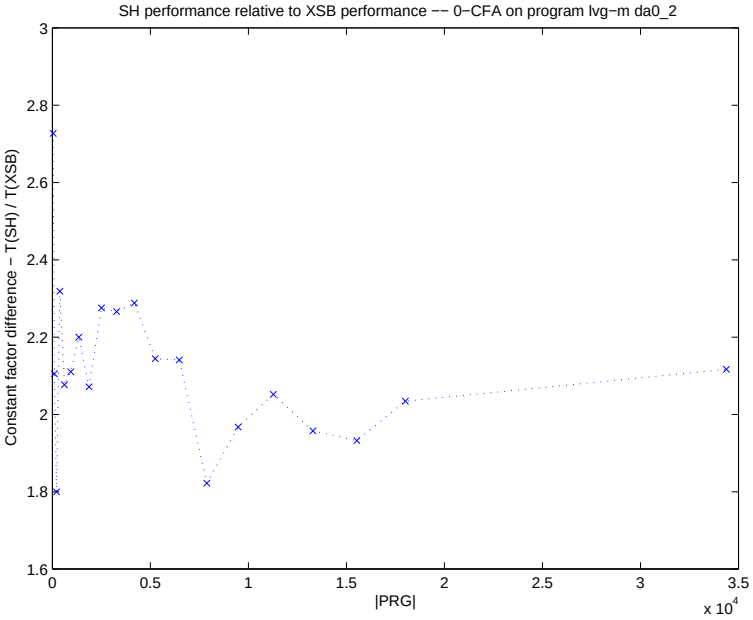
C.1 Individual Solvers





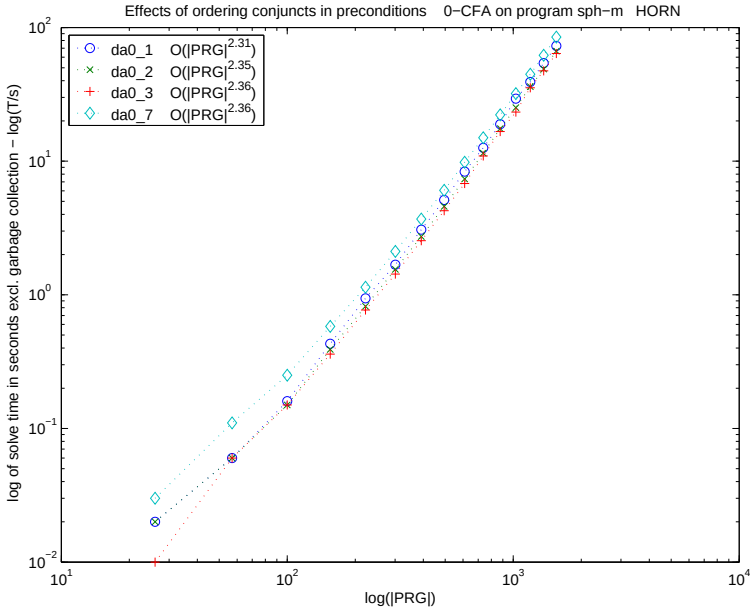
C.2 Solvers Compared

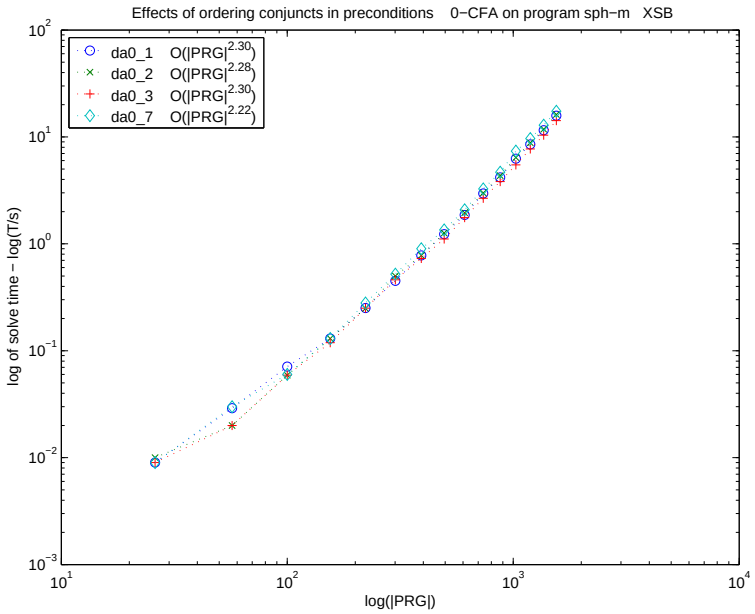




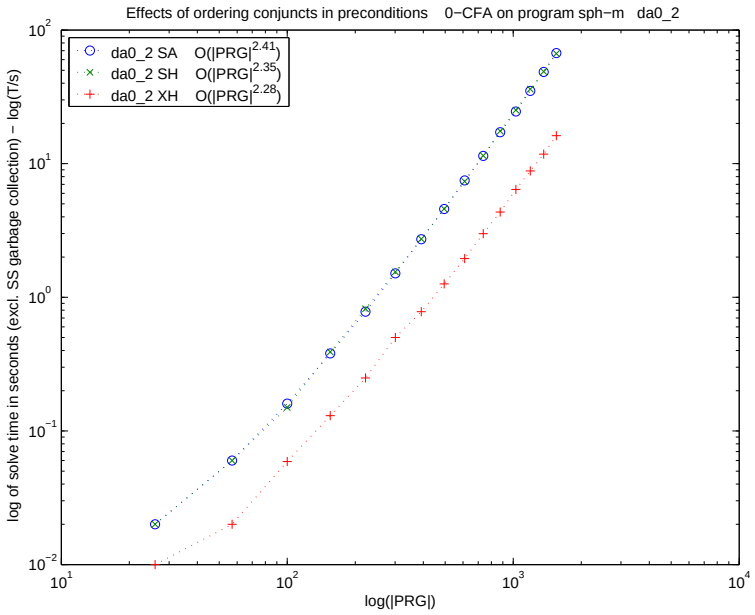
D 0-CFA for the sph(m) Programs

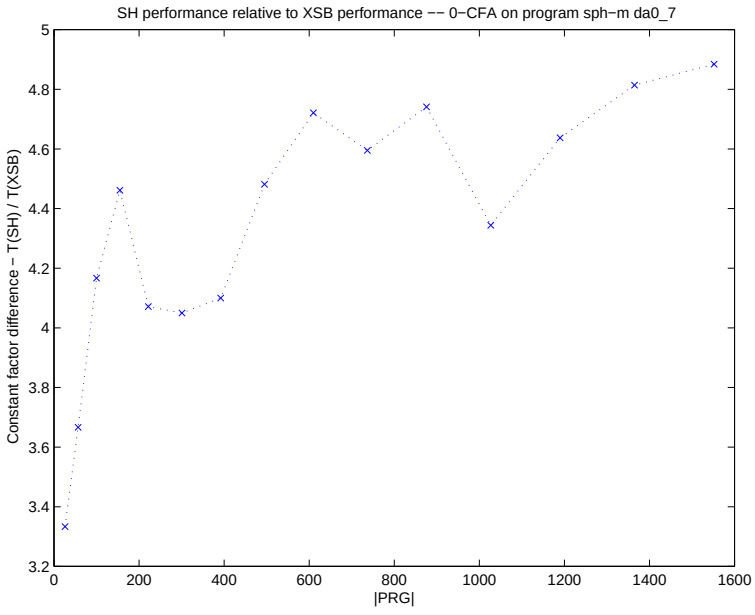
D.1 Individual Solvers





D.2 Solvers Compared





E Closure Condition Examples

E.1 da0_2

$$\forall *, \mu, t_1 : \left[\begin{array}{c} \mathsf{amb}(\mu) = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \mathcal{I}(*, \mu)$$

$$\forall *, \mu, t_1 : \left[\begin{array}{c} \mathsf{in}(\mu) = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*, t_1) \wedge \\ \\ \forall \mu^a, \mu^p, t_2 : \left[\begin{array}{c} \overline{\mathsf{in}}(\mu^a, \mu) = t_2 \wedge \\ \mathcal{I}(\mu, t_2) \wedge \\ \mathcal{I}(\mu^p, \mu) \wedge \\ \mathcal{I}(\mu^a, t_1) \wedge \\ \mathcal{I}(\mu^p, \mu^a) \end{array} \right] \Rightarrow \mathcal{I}(\mu, \mu^a) \end{array} \right]$$

$$\forall *, \mu, t_1 : \left[\begin{array}{c} \mathsf{out}(\mu) = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*, t_1) \wedge \\ \\ \forall \mu^a, \mu^g, t_2 : \left[\begin{array}{c} \overline{\mathsf{out}}(\mu^a, \mu) = t_2 \wedge \\ \mathcal{I}(\mu, t_2) \wedge \\ \mathcal{I}(\mu^g, \mu) \wedge \\ \mathcal{I}(\mu^a, t_1) \wedge \\ \mathcal{I}(\mu, \mu^a) \end{array} \right] \Rightarrow \mathcal{I}(\mu^g, \mu^a) \end{array} \right]$$

$$\forall *, \mu, t_1 : \left[\begin{array}{c} \mathsf{open}(\mu) = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*, t_1) \wedge \\ \\ \forall \mu^p, t_2 : \left[\begin{array}{c} \overline{\mathsf{open}}(\mu^p, \mu) = t_2 \wedge \\ \mathcal{I}(\mu, t_2) \wedge \\ \mathcal{I}(\mu^p, \mu) \wedge \\ \mathcal{I}(\mu^p, t_1) \end{array} \right] \Rightarrow \forall u_1 : \left[\begin{array}{c} \mathcal{I}(\mu, u_1) \end{array} \right] \Rightarrow \mathcal{I}(\mu^p, u_1) \end{array} \right]$$

$$\forall *, \mu, \mu', t_1 : \left[\begin{array}{c} \overline{\mathsf{in}}(\mu, \mu') = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \mathcal{I}(*, t_1)$$

$$\forall *, \mu, \mu', t_1 : \left[\begin{array}{c} \overline{\mathsf{out}}(\mu, \mu') = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \mathcal{I}(*, t_1)$$

$$\forall *, \mu, \mu', t_1 : \left[\begin{array}{c} \overline{\mathsf{open}}(\mu, \mu') = t_1 \wedge \\ \mathsf{PRG}(*, t_1) \end{array} \right] \Rightarrow \mathcal{I}(*, t_1)$$

E.2 da0_7

$$\forall *,\mu,t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \mathsf{amb}(\mu) = t_1 \end{array}\right] \Rightarrow \mathcal{I}(*,\mu)$$

$$\forall *,\mu,t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \mathsf{in}(\mu) = t_1 \end{array}\right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*,t_1)\wedge \\ \\ \forall \mu^a,\mu^p,t_2 : \left[\begin{array}{c} \mathcal{I}(\mu,t_2)\wedge \\ \overline{\mathsf{in}}(\mu^a,\mu) = t_2\wedge \\ \mathcal{I}(\mu^p,\mu)\wedge \\ \mathcal{I}(\mu^a,t_1)\wedge \\ \mathcal{I}(\mu^p,\mu^a) \end{array}\right] \Rightarrow \mathcal{I}(\mu,\mu^a) \end{array}\right]$$

$$\forall *,\mu,t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \mathsf{out}(\mu) = t_1 \end{array}\right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*,t_1)\wedge \\ \\ \forall \mu^a,\mu^g,t_2 : \left[\begin{array}{c} \mathcal{I}(\mu,t_2)\wedge \\ \overline{\mathsf{out}}(\mu^a,\mu) = t_2\wedge \\ \mathcal{I}(\mu^g,\mu)\wedge \\ \mathcal{I}(\mu^a,t_1)\wedge \\ \mathcal{I}(\mu,\mu^a) \end{array}\right] \Rightarrow \mathcal{I}(\mu^g,\mu^a) \end{array}\right]$$

$$\forall *,\mu,t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \mathsf{open}(\mu) = t_1 \end{array}\right] \Rightarrow \left[\begin{array}{l} \mathcal{I}(*,t_1)\wedge \\ \\ \forall \mu^p,t_2 : \left[\begin{array}{c} \mathcal{I}(\mu,t_2)\wedge \\ \overline{\mathsf{open}}(\mu^p,\mu) = t_2\wedge \\ \mathcal{I}(\mu^p,\mu)\wedge \\ \mathcal{I}(\mu^p,t_1) \end{array}\right] \Rightarrow \forall u_1 : \left[\begin{array}{c} \mathcal{I}(\mu,u_1) \end{array}\right] \Rightarrow \mathcal{I}(\mu^p,u_1) \end{array}\right]$$

$$\forall *,\mu,\mu',t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \overline{\mathsf{in}}(\mu',\mu) = t_1 \end{array}\right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall *,\mu,\mu',t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \overline{\mathsf{out}}(\mu',\mu) = t_1 \end{array}\right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall *,\mu,\mu',t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1)\wedge \\ \overline{\mathsf{open}}(\mu',\mu) = t_1 \end{array}\right] \Rightarrow \mathcal{I}(*,t_1)$$

E.3 da0_7 - Horn Expanded

$$\forall \mu, *, t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{amb}(\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,\mu)$$

$$\forall \mu, \mu^a, \mu^p, *, t_1, t_2 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{in}(\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall \mu, \mu^a, \mu^p, *, t_1, t_2 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{in}(\mu) = t_1 \wedge \\ \mathcal{I}(\mu,t_2) \wedge \\ \overline{\mathsf{in}}(\mu^a,\mu) = t_2 \wedge \\ \mathcal{I}(\mu^p,\mu) \wedge \\ \mathcal{I}(\mu^a,t_1) \wedge \\ \mathcal{I}(\mu^p,\mu^a) \end{array} \right] \Rightarrow \mathcal{I}(\mu,\mu^a)$$

$$\forall \mu, \mu^a, \mu^g, *, t_1, t_2 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{out}(\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall \mu, \mu^a, \mu^g, *, t_1, t_2 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{out}(\mu) = t_1 \wedge \\ \mathcal{I}(\mu,t_2) \wedge \\ \overline{\mathsf{out}}(\mu^a,\mu) = t_2 \wedge \\ \mathcal{I}(\mu^g,\mu) \wedge \\ \mathcal{I}(\mu^a,t_1) \wedge \\ \mathcal{I}(\mu,\mu^a) \end{array} \right] \Rightarrow \mathcal{I}(\mu^g,\mu^a)$$

$$\forall \mu, \mu^p, *, t_1, t_2, u_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{open}(\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall \mu, \mu^p, *, t_1, t_2, u_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \mathsf{open}(\mu) = t_1 \wedge \\ \mathcal{I}(\mu,t_2) \wedge \\ \overline{\mathsf{open}}(\mu^p,\mu) = t_2 \wedge \\ \mathcal{I}(\mu^p,\mu) \wedge \\ \mathcal{I}(\mu^p,t_1) \wedge \\ \mathcal{I}(\mu,u_1) \end{array} \right] \Rightarrow \mathcal{I}(\mu^p,u_1)$$

$$\forall \mu, \mu', *, t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \overline{\mathsf{in}}(\mu',\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall \mu, \mu', *, t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \overline{\mathsf{out}}(\mu',\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,t_1)$$

$$\forall \mu, \mu', *, t_1 : \left[\begin{array}{c} \mathsf{PRG}(*,t_1) \wedge \\ \overline{\mathsf{open}}(\mu',\mu) = t_1 \end{array} \right] \Rightarrow \mathcal{I}(*,t_1)$$

F Carmel XSB Runs: demoney

$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
1379	14	3016	0	1.78	8.28	10.41	l-k-s-h-
1379	14	3016	0	1.79	8.29	10.43	l-s-h-k-
1379	14	3016	0	1.80	8.30	10.46	l-h-s-k-
1379	14	3016	0	1.81	8.31	10.43	l-s-k-h-
1379	14	3016	0	1.82	8.28	10.45	l-k-h-s-
1379	14	3016	0	1.83	8.29	10.43	k-l-s-h-
1379	14	3016	0	1.84	8.34	10.53	l-h-k-s-
1379	14	3016	0	1.86	8.28	10.47	k-l-h-s-
1379	14	3016	0	2.49	8.32	11.22	s-k-l-h-
1379	14	3016	0	2.54	8.32	11.21	k-s-h-l-
1379	14	3016	0	2.55	8.30	11.20	s-l-k-h-
1379	14	3016	0	2.56	8.31	11.20	k-s-l-h-
1379	14	3016	0	2.58	8.27	11.17	s-k-h-l-
1379	14	3016	0	2.60	8.30	11.21	s-h-l-k-
1379	14	3016	0	2.60	8.31	11.24	s-h-k-l-
1379	14	3016	0	2.61	8.30	11.25	s-l-h-k-
1379	14	3016	0	3.34	8.30	11.96	h-l-k-s-
1379	14	3016	0	3.36	8.31	11.99	h-l-s-k-
1379	14	3016	0	3.37	8.29	12.00	h-k-l-s-
1379	14	3016	0	3.39	8.30	12.04	h-s-k-l-
1379	14	3016	0	3.42	8.30	12.03	k-h-l-s-
1379	14	3016	0	3.42	8.29	12.03	h-k-s-l-
1379	14	3016	0	3.46	8.31	12.13	h-s-l-k-
1379	14	3016	0	3.48	8.30	12.13	k-h-s-l-

Table 9: XSB results for the demoney program.

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	1379	14	3016	0	2.26	5
SS	1379	14	3016	0	2.17	6
SS	1379	14	3016	0	2.50	11
SS	1379	14	3016	0	2.78	16
SS	1379	14	3016	0	2.89	21
SS	1379	14	3016	0	2.99	26
SS	1379	14	3016	0	3.25	31
SS	1379	14	3016	0	3.45	36

Table 10: Solution size for the demoney program as function of stackdepth.

G Carmel XSB Runs: demoney-local

Solver	$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
SS	1379	14	3016	0	2.14	4.09	10.37	
SH	1379	14	3016	0	2.12	4.59	9.93	
XSB	1379	14	3016	0	1.80	8.25	10.41	l-h-k-s-

where sol(SS) $\subseteq$ sol(XSB) and sol(XSB) $\subseteq$ sol(SS).

Table 11: Results for the demoney-local program.

$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
1379	14	3016	0	1.80	8.24	10.41	l-h-k-s-
1379	14	3016	0	1.80	8.25	10.39	l-s-h-k-
1379	14	3016	0	1.80	8.24	10.38	k-l-s-h-
1379	14	3016	0	1.81	8.25	10.38	l-k-s-h-
1379	14	3016	0	1.81	8.26	10.41	l-s-k-h-
1379	14	3016	0	1.86	8.26	10.44	l-k-h-s-
1379	14	3016	0	1.86	8.27	10.46	k-l-h-s-
1379	14	3016	0	1.87	8.24	10.43	l-h-s-k-
1379	14	3016	0	2.49	8.25	11.12	s-k-h-l-
1379	14	3016	0	2.54	8.26	11.15	s-h-l-k-
1379	14	3016	0	2.55	8.24	11.12	s-h-k-l-
1379	14	3016	0	2.56	8.24	11.15	s-l-h-k-
1379	14	3016	0	2.56	8.25	11.15	s-l-k-h-
1379	14	3016	0	2.57	8.25	11.16	k-s-h-l-
1379	14	3016	0	2.59	8.27	11.17	s-k-l-h-
1379	14	3016	0	2.59	8.26	11.25	k-s-l-h-
1379	14	3016	0	3.31	8.26	11.89	h-l-k-s-
1379	14	3016	0	3.34	8.25	11.93	h-k-l-s-
1379	14	3016	0	3.34	8.27	11.97	k-h-l-s-
1379	14	3016	0	3.36	8.25	11.97	h-l-s-k-
1379	14	3016	0	3.42	8.26	12.04	h-k-s-l-
1379	14	3016	0	3.43	8.25	12.01	k-h-s-l-
1379	14	3016	0	3.45	8.24	12.05	h-s-k-l-
1379	14	3016	0	3.46	8.24	12.04	h-s-l-k-

Table 12: XSB results for the demoney-local program.

Solver	Size of $\tilde{S}$	Size of $\tilde{H}$	Size of $\tilde{L}$	Size of $\tilde{K}$	CPU/s	StackDepth
SS	1379	14	3016	0	2.24	5
SS	1379	14	3016	0	2.20	6
SS	1379	14	3016	0	2.54	11
SS	1379	14	3016	0	2.74	16
SS	1379	14	3016	0	2.84	21
SS	1379	14	3016	0	3.09	26
SS	1379	14	3016	0	3.25	31
SS	1379	14	3016	0	3.51	36

Table 13: Solution size for the demoney-local program as function of stackdepth.

H Carmel XSB Runs: jc212api

$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
1845	33	4662	37	2.40	8.18	10.93	k-l-s-h-
1845	33	4662	37	2.44	8.18	10.97	k-l-h-s-
1845	33	4662	37	2.49	8.18	11.02	k-s-l-h-
1845	33	4662	37	2.51	8.16	11.05	k-h-l-s-
1845	33	4662	37	2.51	8.17	11.01	k-s-h-l-
1845	33	4662	37	2.60	8.18	11.12	h-l-s-k-
1845	33	4662	37	2.60	8.16	11.08	k-h-s-l-
1845	33	4662	37	2.61	8.16	11.11	h-l-k-s-
1845	33	4662	37	2.61	8.18	11.16	l-k-s-h-
1845	33	4662	37	2.61	8.17	11.15	l-s-k-h-
1845	33	4662	37	2.62	8.16	11.14	h-s-k-l-
1845	33	4662	37	2.63	8.16	11.11	h-k-l-s-
1845	33	4662	37	2.63	8.19	11.17	l-s-h-k-
1845	33	4662	37	2.64	8.17	11.15	s-l-k-h-
1845	33	4662	37	2.64	8.18	11.18	s-h-k-l-
1845	33	4662	37	2.64	8.18	11.17	l-h-s-k-
1845	33	4662	37	2.64	8.17	11.16	h-k-s-l-
1845	33	4662	37	2.65	8.17	11.15	s-l-h-k-
1845	33	4662	37	2.65	8.17	11.15	s-k-l-h-
1845	33	4662	37	2.66	8.18	11.17	l-h-k-s-
1845	33	4662	37	2.66	8.19	11.18	s-h-l-k-
1845	33	4662	37	2.67	8.17	11.18	l-k-h-s-
1845	33	4662	37	2.67	8.18	11.18	h-s-l-k-
1845	33	4662	37	2.68	8.17	11.19	s-k-h-l-

Table 14: XSB results for the jc212api program.

Before α-renaming

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	3340	34	6456	37	2.39	5
SS	3686	36	6711	37	2.63	6
SS	4852	36	6711	37	2.85	11
SS	5820	36	6711	37	2.99	16
SS	6663	36	6711	37	3.40	21
SS	7503	36	6711	37	3.42	26
SS	8343	36	6711	37	3.68	31
SS	9183	36	6711	37	4.06	36

Table 15: Solution size for the jc212api program as function of stackdepth.

After α-renaming

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	1807	31	4407	37	2.46	5
SS	1845	33	4662	37	2.53	6
SS	1845	33	4662	37	2.65	11
SS	1845	33	4662	37	2.79	16
SS	1845	33	4662	37	3.12	21
SS	1845	33	4662	37	3.04	26
SS	1845	33	4662	37	3.43	31
SS	1845	33	4662	37	3.54	36

Table 16: Solution size for the jc212api program as function of stackdepth.

I Carmel XSB Runs: API

$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
2251	40	5404	39	2.75	8.30	11.44	k-h-s-l-
2251	40	5404	39	2.76	8.32	11.41	k-h-l-s-
2251	40	5404	39	2.78	8.29	11.44	k-s-h-l-
2251	40	5404	39	2.83	8.30	11.51	h-l-k-s-
2251	40	5404	39	2.83	8.31	11.49	h-k-l-s-
2251	40	5404	39	2.83	8.47	11.64	k-s-l-h-
2251	40	5404	39	2.84	8.30	11.48	h-l-s-k-
2251	40	5404	39	2.88	8.31	11.52	h-s-k-l-
2251	40	5404	39	2.91	8.54	11.78	h-s-l-k-
2251	40	5404	39	2.91	8.31	11.56	h-k-s-l-
2251	40	5404	39	2.95	8.31	11.72	s-l-h-k-
2251	40	5404	39	2.95	8.32	11.63	s-k-l-h-
2251	40	5404	39	2.97	8.31	11.65	s-l-k-h-
2251	40	5404	39	2.99	8.43	11.76	k-l-s-h-
2251	40	5404	39	3.00	8.32	11.68	s-h-l-k-
2251	40	5404	39	3.01	8.31	11.68	s-k-h-l-
2251	40	5404	39	3.02	8.43	11.79	s-h-k-l-
2251	40	5404	39	3.03	8.32	11.69	k-l-h-s-
2251	40	5404	39	3.40	8.31	12.08	l-s-k-h-
2251	40	5404	39	3.42	8.31	12.09	l-s-h-k-
2251	40	5404	39	3.42	8.30	12.07	l-k-s-h-
2251	40	5404	39	3.44	8.31	12.12	l-h-s-k-
2251	40	5404	39	3.45	9.57	13.42	l-h-k-s-
2251	40	5404	39	3.47	8.30	12.10	l-k-h-s-

Table 17: XSB results for the API program.

Before α-renaming

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	3728	41	7150	39	2.70	5
SS	4074	43	7405	39	2.80	6
SS	5240	43	7405	39	3.02	11
SS	6208	43	7405	39	3.21	16
SS	7051	43	7405	39	3.47	21
SS	7891	43	7405	39	3.69	26
SS	8731	43	7405	39	3.95	31
SS	9571	43	7405	39	4.22	36

Table 18: Solution size for the API program as function of stackdepth.

After α-renaming

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	2213	38	5149	39	2.65	5
SS	2251	40	5404	39	2.68	6
SS	2251	40	5404	39	2.95	11
SS	2251	40	5404	39	2.99	16
SS	2251	40	5404	39	3.16	21
SS	2251	40	5404	39	3.47	26
SS	2251	40	5404	39	3.73	31
SS	2251	40	5404	39	3.82	36

Table 19: Solution size for the API program as function of stackdepth.

J Carmel XSB Runs: demoney-impl-api

$\mid \hat{S} \mid$	$\mid \hat{H} \mid$	$\mid \hat{L} \mid$	$\mid \hat{K} \mid$	CPU/s	Init/s	Time/s	Order
4262	75	11103	39	13.37	16.22	30.24	s-h-k-l-
4262	75	11103	39	13.38	16.22	30.36	s-h-l-k-
4262	75	11103	39	13.42	16.20	30.25	s-l-k-h-
4262	75	11103	39	13.46	16.25	30.34	s-k-h-l-
4262	75	11103	39	13.47	16.21	30.31	s-l-h-k-
4262	75	11103	39	13.56	16.22	30.38	s-k-l-h-
4262	75	11103	39	13.90	16.19	30.75	h-l-s-k-
4262	75	11103	39	13.92	16.21	30.74	h-k-l-s-
4262	75	11103	39	13.95	16.21	30.81	h-l-k-s-
4262	75	11103	39	14.07	16.55	31.29	h-s-k-l-
4262	75	11103	39	14.11	16.20	30.93	h-k-s-l-
4262	75	11103	39	14.12	16.20	30.93	h-s-l-k-
4262	75	11103	39	15.25	16.18	32.12	l-s-h-k-
4262	75	11103	39	15.31	16.20	32.14	l-s-k-h-
4262	75	11103	39	15.32	16.20	32.17	l-k-s-h-
4262	75	11103	39	15.39	16.25	32.28	l-h-s-k-
4262	75	11103	39	15.41	16.21	32.25	l-k-h-s-
4262	75	11103	39	15.43	16.19	32.24	l-h-k-s-
4262	75	11103	39	19.51	16.21	36.43	k-l-s-h-
4262	75	11103	39	19.69	16.20	36.53	k-l-h-s-
4262	75	11103	39	19.96	16.21	36.84	k-s-l-h-
4262	75	11103	39	20.01	16.21	36.89	k-s-h-l-
4262	75	11103	39	20.25	16.21	37.17	k-h-l-s-
4262	75	11103	39	20.47	16.21	37.30	k-h-s-l-

Table 20: XSB results for the demoney-impl-api program.

Before α-renaming

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	6806	84	13857	39	6.70	5
SS	6806	84	13857	39	6.53	6
SS	7870	84	13857	39	6.72	11
SS	8997	84	13857	39	7.30	16
SS	9980	84	13857	39	7.80	21
SS	10960	84	13857	39	8.01	26
SS	11940	84	13857	39	8.54	31
SS	12920	84	13857	39	9.10	36

Table 21: Solution size for the demoney-impl-api program as function of stackdepth.

After α-renaming

Solver	Size of $\hat{S}$	Size of $\hat{H}$	Size of $\hat{L}$	Size of $\hat{K}$	CPU/s	StackDepth
SS	4262	75	11103	39	6.42	5
SS	4262	75	11103	39	6.33	6
SS	4262	75	11103	39	6.89	11
SS	4262	75	11103	39	7.11	16
SS	4262	75	11103	39	7.63	21
SS	4262	75	11103	39	8.06	26
SS	4262	75	11103	39	8.24	31
SS	4262	75	11103	39	8.61	36

Table 22: Solution size for the demoney-impl-api program as function of stackdepth.