Kent Academic Repository

Robbins, Ed, Howe, Jacob M. and King, Andy (2015) *Theory Propagation and Reification.* Science of Computer Programming, 111 (1). pp. 3-22. ISSN 0167-6423.

Downloaded from

https://kar.kent.ac.uk/37600/ The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.1016/j.scico.2014.05.013

This document version

Pre-print

DOI for this version

Licence for this version

UNSPECIFIED

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Theory Propagation and Reification

Ed Robbins[a,*], Jacob M. Howe[b], Andy King[a]

[a] *School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK*
[b] *Department of Computer Science, City University London, London, EC1V 0HB, UK*

Abstract

SAT Modulo Theories (SMT) is the problem of determining the satisfiability of a formula in which constraints, drawn from a given constraint theory T, are composed with logical connectives. The DPLL(T) approach to SMT has risen to prominence as a technique for solving these quantifier-free problems. The key idea in DPLL(T) is to couple unit propagation in the propositional part of the problem with theory propagation in the constraint component. In this paper it is demonstrated how reification provides a natural way for orchestrating this in the setting of logic programming. This allows an elegant implementation of DPLL(T) solvers in Prolog. The work is motivated by a problem in reverse engineering, that of type recovery from binaries. The solution to this problem requires an SMT solver where the theory is that of rational-tree constraints, a theory not supported in off-the-shelf SMT solvers, but realised as unification in Prolog systems. The approach is also illustrated with SMT solvers for linear constraints and integer difference constraints. The rational-tree solver is benchmarked against a number of type recovery problems, and compared against a lazy-basic SMT solver built on PicoSAT, while the integer difference logic solver is benchmarked against CVC3 and CVC4, both of which are implemented in C++.

*Corresponding author

Email address: `er209@kent.ac.uk` (Ed Robbins)

The PPDP version of the paper, unchanged apart from reformatted in SCP style, is available from `http://www.cs.kent.ac.uk/people/staff/amk/theory-propagation-scp.pdf`

New material:. The paper includes a number of sections of new material over and above the PPDP version:

- A correctness proof of the core algorithm (an extended section 4.3);
- An extension of the work to difference logic (new section 7);
- Benchmarking difference logic against CVC3 and CVC4 which is reflected in the new structure for the experimental results (new section 8.2) and the discussion (new section 9.2).

---

1. Introduction

DPLL-based SAT solvers have advanced to the point where they can rapidly decide the satisfiability of structured problems that involve thousands of variables. SAT Modulo Theories (SMT) seeks to extend these ideas beyond propositional formulae to formulae that are constructed from logical connectives that combine constraints drawn from a given underlying theory. This section introduces the motivating problem of type recovery and explains why it leads to work on theory propagation in a Prolog SMT solver.

1.1. Type recovery with SMT

The current work is motivated by reverse engineering and the problem of type recovery from binaries. Reversing executable code is of increasing relevance for a range of applications:

- Exposing flaws and vulnerabilities in commercial software, especially prior to deployment in government or industry [13, 19];
- Reuse of legacy software without source code for guaranteed compliance with hardware IO or timing behaviour, for example, for hardware drivers [11] or control systems [8];
- Understanding the operation of, and threat posed by, viruses and other malicious code by anti-virus companies [50].

An important problem in reverse engineering is that of type recovery [43]. A fragment of binary code will almost certainly have multiple source code equivalents, will contain a variety of complex addressing schemes, and during

compilation will have lost most, if not all, of the type information explicit in the original source code. Additionally, container-like entities, analogous to high level source code variables and objects, cannot be readily extracted from binary code. The recovery of variables and their types is an essential component of reverse engineering, which makes understanding the semantics of the program considerably easier.

This paper observes that type recovery can be formulated as an SMT problem over rational-trees, a theory that in the context of type checking is referred to as circular unification [38]. Circular unification allows recursive types to be discovered in which a type variable can be unified with a term containing it. The use of rational-trees for type inference is not a new idea [38], but its application to the recovery of recursive types from an executable is far from straightforward because each instruction can be assigned many different types. Many SMT solvers include the theory of equality logic over uninterpreted functors [31, 45] which is strictly weaker than circular unification and cannot capture recursive types. Unfortunately the theory of rational-trees is not currently supported in any off-the-shelf SMT solver, hence this investigation into how to build a solver.

1.2. SMT solving with lazy-basic

One straightforward approach to SMT solving is to apply the so-called lazy-basic technique which decouples SAT solving from theory solving. To illustrate, consider the SMT formula $f = (x \leq -1 \vee -x \leq -1) \wedge (y \leq -1 \vee -y \leq -1)$ and the SAT formula $g = (p \vee q) \wedge (r \vee s)$ that corresponds to its propositional skeleton. In the skeleton, the propositional variables p, q, r and s, respectively, indicate whether the theory constraints $(x \leq -1)$, $(-x \leq -1)$, $(y \leq -1)$ and $(-y \leq -1)$ hold. In this approach, a model is found for $(p \vee q) \wedge (r \vee s)$, for instance, $\{p \mapsto true, q \mapsto true, r \mapsto true, s \mapsto false\}$. Then, from the model, a conjunction of theory constraints $(x \leq -1) \wedge (-x \leq -1) \wedge (y \leq -1) \wedge \neg(-y \leq -1)$ is constructed, with the polarity of the constraints reflecting the truth assignment. This conjunction is then tested for satisfiability in the theory component. In this case it is unsatisfiable, which triggers a diagnostic stage. This amounts to finding a conjunct, in this case $(x \leq -1) \wedge (-x \leq -1)$, which is also unsatisfiable, that identifies a source of the inconsistency. From this conjunct, a blocking clause $(\neg p \vee \neg q)$ is added to g to give g' which ensures that conflict between the theory constraints is never encountered again. Then, solving the augmented propositional formula g' might, for example, yield the

model $\{p \mapsto false, q \mapsto true, r \mapsto true, s \mapsto true\}$, from which a second clause $(\neg r \vee \neg s)$ is added to g'. Any model subsequently found, for instance, $\{p \mapsto false, q \mapsto true, r \mapsto true, s \mapsto false\}$, will give a conjunction that is satisfiable in the theory component, thereby solving the SMT problem.

The lazy-basic approach is particularly attractive when combining an existing SAT solver with an existing decision procedure, for instance, a solver provided by a constraint library. By using a foreign language interface a SAT solver can be invoked from Prolog [12] and a constraint library can be used to check satisfiability of the conjunction of theory constraints. A layer of code can then be added to diagnose the source of any inconsistency. This provides a simple way to construct an SMT solver that compares very favourably with the coding effort required to integrate a new theory into an existing open source SMT solver. The latter is normally a major undertaking and often can only be achieved in conjunction with the expert who is responsible for maintaining the solver. Furthermore, few open source solvers are actively maintained. Thus, although one might expect implementing a new theory to be merely an engineering task, it is actually far from straightforward.

Prolog has rich support for implementing decision procedures for theories, for instance, attributed variables [20, 21]. (Attributed variables provide an interface between Prolog and a constraint solver by permitting logical variables to be associated with state, for instance, the range of values that a variable can possibly assume.) Several theories come prepackaged with many Prolog systems. This raises the questions of how to best integrate a theory solver with a SAT solver, and how powerful an SMT solver written in a declarative language can actually be. This motivates further study of the coupling between the theory and the propositional component of the SAT solver which goes beyond the lazy-basic approach, to the roots of logic programming itself.

The equation Algorithm = Logic + Control [33] expresses the idea that in logic programming algorithm design can be decoupled into two separate steps: specifying the logic of the problem, classically as Horn clauses, and orchestrating control of the sub-goals. The problem of satisfying a SAT formula is conceptually one of synchronising activity between a collection of processes where each process checks the satisfiability of a single clause. Therefore it is perhaps no surprise that control primitives such as delay declarations [44] can be used to succinctly specify the watched literal technique [42]. In this technique, a process is set up to monitor two variables of each clause. To illustrate, consider the clause $(x \vee y \vee \neg z)$. The process for this clause will

suspend on two of its variables, say x and y, until one of them is bound to a truth-value. Suppose x is bound. If x is bound to $true$ then the clause is satisfied, and the process terminates; if x is bound to $false$, then the process suspends until either y or z is bound. Suppose z is subsequently bound, either by another process or by labelling. If z is $true$ then y is bound to $true$ since otherwise the clause is not satisfied; if z is $false$ then the clause is satisfied and the process closes down without inferring any value for y. Note that in these steps the process only waits on two variables at any one time. Unit propagation is at the heart of SAT solving and when implemented by watched literals combined with backtracking, the resulting solver is efficient enough to solve some non-trivial propositional formulae [22, 23, 25]. In addition to issues of performance the correctness of this approach has been examined [17]. To summarise, Prolog not only provides constraint libraries, but also the facility to implement a succinct SAT solver [25]. The resulting solver can be regarded as a glass box, as opposed to a black one, which allows a solver to be extended to support, among other things, new theories and theory propagation.

1.3. SMT solving with theory propagation

The lazy-basic approach to SMT alternates between SAT solving and checking whether a conjunction of theory constraints is satisfiable which, though having conceptual and implementation advantages, is potentially inefficient. With a glass box solver it is possible to refine this interaction by applying theory propagation. In theory propagation, the SAT solving and theory checking are interleaved. The solver not only checks the satisfiability of a conjunction of theory constraints, but decides whether a conjunction of some constraints entails or disentails others. Returning to the earlier example, observe that $(x \leq -1) \wedge (-x \leq -1)$ is unsatisfiable, hence for the partial assignment $\{p \mapsto true\}$ it follows that $(x \leq -1)$ holds in the theory component, therefore $(-x \leq -1)$ is disentailed and the assignment can be extended to $\{p \mapsto true, q \mapsto false\}$. Theory propagation is essentially the coordination problem of scheduling unit propagation with the simultaneous checking of whether theory constraints are entailed or disentailed. This paper shows how this synchronisation can be realised straightforwardly in Prolog, again using control primitives. The resulting solver is capable of solving some non-trivial problems and performs well against an off-the-shelf DPLL(T) solver on integer difference logic benchmarks, whilst on rational-trees it outperforms

an SMT solver constructed from PicoSAT [6] and a Prolog coded theory solver using the lazy-basic approach.

1.4. Contributions

This paper revises and extends [49] and shows how to integrate theory propagation and unit propagation in Prolog using reification and thereby realise an SMT solver in Prolog. Reification is a constraint handling mechanism in which a constraint is augmented with a Boolean variable that indicates whether the constraint is entailed (implied by the store) or disentailed (is inconsistent with the store). Building on this mechanism, the paper makes the following contributions:

- A framework for using reification as a mechanism to realise theory propagation is presented. The idea is simple in hindsight and can be realised straightforwardly in Prolog. The simplicity of the code contrasts with the investment required to integrate a theory into an existing open source SMT solver.
- This framework is realised for three theories:
  - The first theory is that of rational-trees [39], where the control provided by block and when-declarations can realise reification. Efficient rational-tree unification [27] is integral to many Prolog systems, hence the theory part of the solver is provided essentially for free.
  - The second theory is that of quantifier-free linear real arithmetic, where CLP($\mathcal{R}$) provides a decision procedure for the theory part of the solver; reification is achieved using a combination of delay declarations and entailment checking.
  - The third theory is that of quantifier-free integer difference logic, with the decision procedure for the theory coded in Prolog and theory propagation realised again using entailment checking and delay declarations.
- Quantifier-free integer difference logic is a theory that is a common part of SMT packages. As a strength test, the Prolog-based solver is benchmarked against a popular open-source SMT solver, CVC, using standard SMT-LIB benchmarks.

- Theory propagation for rational-trees provides the key motivation for the paper. Standard SMT packages do not include the theory of rational-trees, but SMT problems over rational-trees arise in reverse engineering, in particular type recovery. It is demonstrated that an elegant Prolog-based solver is capable of recovering types for a range of binaries. It is also shown how the failed literal technique [36] is simply realised in Prolog to optimise the search. The solver is benchmarked on these type recovery problems and also compared against an SMT solver constructed from PicoSAT using the lazy-basic approach.

- Cutting through all of these contributions, the paper also argues that SMT has a role in type recovery, indeed an SMT formula is a natural medium for expressing the disjunctive nature of the types that arise in reverse engineering.

2. Motivation: Application in Type Recovery

During compilation code is translated to low level operations on registers and memory addresses, and all type information is lost. When source code is not available, type information is of great use to reverse engineers in determining the operation of a program, and tooling for recovery of high level types is of significant utility. The problem is hard, since the typing of most assembly instructions can be interpreted in multiple ways, and progress on the problem has been comparatively slow [3, 10, 35, 37, 43, 48], stopping short of recovering recursive types.

Consider the problem of inferring types for the registers in the following x86 assembly code function for summing the elements in a linked list of type **struct** A {**int** value; **struct** A ∗**next**}. Note this function is based on Mycroft's Register Transfer Language (RTL) example [43].

```
1            mov edx, [esp+0x4]
2            mov eax, 0x0
3   loop:    test edx, edx
4            jz end
5            add eax, [edx]
6            mov edx, [edx+0x4]
7            jmp loop
8   end:     ret
```

The function is simple: first **edx** is set to point at the first list item (from the argument carried at [**esp** + **0x4**]) and **eax**, the accumulator, is initialised to 0 (lines 1 and 2). In the loop body the **value** of the item is added to **eax** (line 5) and **edx** is set to point to the next item by dereferencing the **next** field from [**edx** + **0x4**] (line 6). This repeats until a NULL pointer is found by the test on line 3, whereupon execution jumps to **end** and the function returns.

Before typing the function, indirect addressing is simplified by introducing new operations on fresh intermediate variables. This reduction ensures that indirect addressing only ever occurs on **mov** instructions, thus simplifying the constraints on all other instructions. Registers are then broken into live ranges by transforming into Single Static Assignment (SSA) form. This gives each variable a new index whenever it is written to, and joins variables at control flow merge points with ϕ functions [14]. The listing below shows the result of applying these transformations:

```
1                mov A1, esp0
2                add A2, 0x4
3                mov edx1, [A2]
4                mov eax1, 0x0
5    loop:       mov (eax2, edx2),
                     φ((eax1, edx1), (eax3, edx3))
6                test edx2, edx2
7                jz end
8                mov B1, [edx2]
9                add eax3, B1
10               mov C1, edx2
11               add C2, 0x4
12               mov edx3, [C2]
13               jmp loop
14   end:        ret
```

Rational-tree expressions [26], constraints describing unification of terms and type variables, are now derived for each instruction. These are similar to the disjunctive constraints described by [43] for RTL, but include a memory model that tracks pointer manipulation by representing memory in 'pointed to' locations as a 3-tuple. The type of the specific location being pointed

to is the middle element, the first element is a list of types for the bytes preceding the location, and the last the types for the bytes succeeding. The lists are open, as indicated by the ellipsis (...), since the areas of memory extending to either side are unknown. For example, consider **add** on line 11. This gives rise to two constraints, one for each possible meaning of the code:

$$
\begin{array}{c}
(T_{C_2} = \text{basic}(_, \text{int}, 4) \wedge T_{C_1} = T_{C_2}) \\
\vee \left(\begin{array}{l} T_{C_1} = \text{ptr}(\langle[\ldots], \beta_0, [\beta_1, \beta_2, \beta_3, \beta_4, \ldots]\rangle) \wedge \\ T_{C_2} = \text{ptr}(\langle[\ldots, \beta_0, \beta_1, \beta_2, \beta_3], \beta_4, [\ldots]\rangle) \end{array} \right)
\end{array}
$$

The first clause of the disjunction states that $\mathbf{C_2}$ is of basic type, specifically a four byte integer (derived from the register size) with unknown signedness (as indicated by a sign parameter that is an uninstantiated variable), the result of adding 4 to $\mathbf{C_1}$, which has the same type. This is disjoint from the second clause, that asserts that $\mathbf{C_1}$ is a pointer to an unknown type β_0, whose address is incremented by 4 by the **add** operation so that its new instance, $\mathbf{C_2}$, points to another location of type β_4. Observe how T_{C_1} prescribes types of objects that follow the object of type β_0 in memory whereas T_{C_2} details types of objects that precede the object of type β_4. If further information is later added to T_{C_2} due to unification it will propagate into T_{C_1}, and vice-versa, thus aggregate types analogous to C structs are derived.

The table below shows all constraints generated for the program. Note that some type variables have been relaxed to _, indicating an uninstantiated variable, so as to simplify the presentation of the types. The complete problem is described by the conjunction of these constraints. Type recovery then amounts to solving the constraints such that the type equations remain consistent, whilst also ensuring that the propositional skeleton of the problem is satisfied.

Line	Generated Constraints
1	$T_{A_1} = T_{esp_0}$
2	$(T_{A_2} = \text{basic}(_, \text{int}, 4) \wedge T_{A_1} = T_{A_2}) \vee$ $\left(\begin{array}{c} T_{A_1} = \text{ptr}(\langle[\ldots], \alpha_0, [_, _, _, \alpha_1, \ldots]\rangle) \wedge \\ T_{A_2} = \text{ptr}(\langle[\ldots, \alpha_0, _, _, _], \alpha_1, [\ldots]\rangle) \end{array} \right)$
3	$T_{A_2} = \text{ptr}(\langle[\ldots], T_{edx_1}, [_, _, _, \ldots]\rangle)$
4	$T_{eax_1} = \text{basic}(_, \text{int}, 4) \vee T_{eax_1} = \text{ptr}(\langle[\ldots], \alpha_2, [\ldots]\rangle)$
5	$(T_{eax_2} = T_{eax_1} \wedge T_{edx_2} = T_{edx_1}) \wedge (T_{eax_2} = T_{eax_3} \wedge T_{edx_2} = T_{edx_3})$
8	$T_{edx_2} = \text{ptr}(\langle[\ldots], T_{B_1}, [\ldots]\rangle)$
9	$\left(\begin{array}{c} T_{eax_3} = \text{basic}(_,\text{int},4) \wedge \\ T_{eax_2} = T_{eax_3} \wedge T_{B_1} = T_{eax_3} \end{array} \right) \vee$ $\left(\begin{array}{c} T_{eax_3} = \text{ptr}(\langle[\ldots], \alpha_3, [\ldots]\rangle) \wedge \\ T_{eax_2} = \text{ptr}(\langle[\ldots], \alpha_4, [\ldots]\rangle) \wedge \\ T_{B_1} = \text{basic}(_,\text{int},4)) \end{array} \right) \vee$ $\left(\begin{array}{c} T_{eax_3} = \text{ptr}(\langle[\ldots], \alpha_5, [\ldots]\rangle) \wedge \\ T_{eax_2} = \text{basic}(_,\text{int},4) \wedge \\ T_{B_1} = \text{ptr}(\langle[\ldots], \alpha_6, [\ldots]\rangle) \end{array} \right)$
10	$T_{edx_2} = T_{C_1}$
11	$(T_{C_2} = \text{basic}(_, \text{int}, 4) \wedge T_{C_1} = T_{C_2}) \vee$ $\left(\begin{array}{c} T_{C_1} = \text{ptr}(\langle[\ldots], \alpha_7, [_, _, _, \alpha_8, \ldots]\rangle) \wedge \\ T_{C_2} = \text{ptr}(\langle[\ldots, \alpha_7, _, _, _], \alpha_8, [\ldots]\rangle) \end{array} \right)$
12	$T_{C_2} = \text{ptr}(\langle[\ldots], T_{edx_3}, [_, _, _, \ldots]\rangle)$

For the register corresponding to **struct** A, constraint solving will derive a recursive type:

$$T_{edx_1} = \text{ptr}(\langle[\ldots], \text{basic}(_, \text{int}, 4), [_, _, _, T_{edx_1}, _, _, _, \ldots]\rangle)$$

which requires rational-tree unification. Note that as T_{edx_1} is a pointer, it has a size of four bytes, hence the 3 underscores that follow T_{edx_1} which correspond to 3 bytes of padding in our representation.

Observe that there may be multiple solutions; in fact the problem outlined above has two solutions, which differ in typing $\mathbf{eax}_1$, $\mathbf{eax}_2$ and $\mathbf{eax}_3$. The first correctly infers that they are (like B_1) integers of size 4 bytes, while the second defines them as pointers to an unknown type, $\text{ptr}(\langle[\ldots], \alpha_5, [\ldots]\rangle)$.

Both solutions have the following typings in common:

$$T_{B_1} = \text{basic}(_,\text{int},4)$$
$$T_{edx_1} = T_{edx_2} = T_{edx_3} = T_{C_1} = \text{ptr}(\langle[...], \text{basic}(_, \text{ int, } 4), [_, _, _, T_{edx_1}, _, _, _, ...]\rangle)$$
$$T_{C_2} = \text{ptr}(\langle[..., \text{basic}(_,\text{int},4), _, _, _], T_{edx_1}, [_, _, _, ...]\rangle)$$
$$T_{A_2} = \text{ptr}(\langle[..., \alpha_0, _, _, _], T_{edx_1}, [_, _, _, ...]\rangle)$$
$$T_{A_1} = T_{esp_0} = \text{ptr}(\langle[...], \alpha_0, [_, _, _, T_{edx_1}, _, _, _, ...]\rangle)$$

The second solution is equivalent to typing **eax** as **void**$*$ and performing addition using pointer arithmetic. In the wider context of a program, this solution is removed by constraints derived from the *main()* function.

3. SMT and Theory Propagation

3.1. SAT solving and unit propagation

The Boolean satisfiability problem (SAT) is the problem of determining whether for a given Boolean formula, there is a truth assignment to the variables of the formula under which the formula evaluates to *true*. Most recent SAT solvers are based on the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [15] with watched literals [42]; this includes the solver in [24] that this paper extends.

At the heart of the DPLL approach is unit propagation. Let f be a propositional formula in CNF over a set of propositional variables X. Let $\theta : X \to \{\mathit{true}, \mathit{false}\}$ be a partial (truth) function. Unit propagation examines each clause in f to deduce a truth assignment θ' that extends θ and necessarily holds for f to be satisfiable. For example, suppose $f = (\neg x \vee z) \wedge (u \vee \neg v \vee w) \wedge (\neg w \vee y \vee \neg z)$ so that $X = \{u, v, w, x, y, z\}$ and θ is the partial function $\theta = \{x \mapsto \mathit{true}, y \mapsto \mathit{false}\}$. In this instance for the clause $(\neg x \vee z)$ to be satisfiable, hence f as a whole, it is necessary that $z \mapsto \mathit{true}$. Moreover, for $(\neg w \vee y \vee \neg z)$ to be satisfiable, it follows that $w \mapsto \mathit{false}$. The satisfiability of $(u \vee \neg v \vee w)$ depends on two unknowns, u and v, hence no further information can be deduced from this clause. Therefore $\theta' = \theta \cup \{w \mapsto \mathit{false}, z \mapsto \mathit{true}\}$.

Searching for a satisfying assignment proceeds as follows: starting from an empty truth function θ, an unassigned variable occurring in f, x, is selected and $x \mapsto \mathit{true}$ is added to θ. Unit propagation extends θ until either no further propagation is possible or a contradiction is established. In the first

case, if all clauses are satisfied then f is satisfied, else another unassigned variable is selected. In the second case, $x \mapsto false$ is added to θ; if this fails search backtracks to a previous assignment. Further details can be found in [24, 56].

3.2. SMT solving, the lazy-basic approach

SAT modulo theories (SMT) gives a general scheme for determining the satisfiability of problems consisting of a formula over atomic constraints in some theory T, whose set of literals is denoted Σ [46, 54]. The scheme separates the propositional skeleton – the logical structure of combinations of theory literals – and the meaning of the literals. A bijective encoder mapping $e : \Sigma \to X$ associates each literal with a unique propositional variable. Then the encoder mapping e is lifted to theory formulae, using $e(\phi)$ to denote the propositional skeleton of a theory formula ϕ.

Consider the theory of quantifier-free linear real arithmetic where the constants are numbers, the functors are interpreted as addition and subtraction, and the predicates include equality, disequality and both strict and non-strict inequalities. The problem of checking the entailment $(a < b) \wedge (a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1) \models (a + b = 1)$ amounts to determining that the theory formula $\phi = (a < b) \wedge (a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1) \wedge \neg(a + b = 1)$ is not satisfiable. For this problem, the set of literals is $\Sigma = \{a < b, \ldots, a + b = 1\}$. Suppose, in addition, that the encoder mapping is defined:

$$\begin{array}{lll} e(a < b) = x, & e(a = 0) = y, & e(a = 1) = z, \\ e(b = 0) = u, & e(b = 1) = v, & e(a + b = 1) = w \end{array}$$

Then the propositional skeleton of ϕ, given e, is $e(\phi) = x \wedge (y \vee z) \wedge (u \vee v) \wedge \neg w$. A SAT solver gives a truth assignment θ satisfying the propositional skeleton. From this, a conjunction of theory literals, $\hat{Th}_\Sigma(\theta, e)$ is constructed. The conjunction contains the literal ℓ if $\theta(e(\ell)) = true$ and $\neg\ell$ if $\theta(e(\ell)) = false$. The subscript will be omitted when Σ refers to all literals in a problem. This problem is passed to a solver for the theory that can determine satisfiability of conjunctions of constraints. Either satisfiability or unsatisfiability is determined, in the latter case the SAT solver is asked for further satisfying truth assignments. This formulation is known as the lazy-basic approach and details on its Prolog implementation can be found in [25].

3.3. SMT, the DPLL(T) approach

The approach detailed in the previous section finds complete satisfying assignments to the SAT problem given by the propositional skeleton before

```
function DPLL(T)(f: CNF formula, θ : truth assignment, e : Σ → X)
begin
    (θ3, res) := propagate(f, θ, e, ∅);
    if (is-satisfied(f, θ3)) then
        return θ3;
    else if (res = ⊥) then
        return ⊥;
    else
        x := choose-free-variable(f, θ3);
        (θ4, res) := DPLL(T)(f, θ3 ∪ {x ↦ true}, e);
        if (res = ⊤) then
            return θ4;
        else
            return DPLL(T)(f, θ3 ∪ {x ↦ false}, e);
        endif
    endif
end
```

```
function propagate(f: CNF formula, θ : truth assignment,
                   e : Σ → X, D : set of theory literals)
begin
    θ1 := θ ∪ {e(ℓ) ↦ true | ℓ ∈ D ∩ Σ} ∪ {e(ℓ) ↦ false | ¬ℓ ∈ D ∧ ℓ ∈ Σ};
    θ2 := θ1∪ unit-propagation(f,θ1);
    ⟨D, res⟩ := deduction(T̂h(θ2, e));
    if (D = ∅ ∨ res = ⊥)
        return (θ2, res);
    else
        return propagate(f, θ2, e, D);
    endif
end
```

Figure 1: Recursive formulation of the DPLL(T) algorithm

computing the satisfiability of the theory problem $\hat{Th}(\theta, e)$. Another approach is to couple the SAT problem and the theory problem more tightly by determining constraints entailed by the theory and propagating the bindings back into the SAT problem. This is known as theory propagation and

is encapsulated in the DPLL(T) approach. Figure 1 gives a recursive formulation of DPLL(T) derived from Algorithm 11.2.3 of [34]. A more general formulation of DPLL(T) might replace lines (11)-(15) with a conflict analysis step that would encapsulate not just the approach presented, but also backjumping and clause learning heuristics. However, the key component of DPLL(T) is the interleaving of unit and theory propagation and the choice of conflict analysis is an orthogonal issue. The instantiation to chronological backtracking presented in Figure 1 was chosen to match the implementation work.

The first argument to the function DPLL(T) is a Boolean formula f, its second a partial truth assignment, θ, and its third an encoder mapping, e. In the initial call, f is the propositional skeleton of the input problem, $e(\phi)$, and θ is empty. DPLL(T) returns a truth assignment if the problem is satisfiable and constant $\perp$ otherwise.

The call to propagate is the key operation. The function returns a pair consisting of a truth assignment and res taking value $\top$ or $\perp$ indicating the satisfiability of f and $\hat{Th}(\theta, e)$. The fourth argument to propagate is a set of theory literals, D, and the function begins by extending the truth assignment by assigning propositional variables identified by the encoder mapping. Next, unit propagation as described in section 3.1 is applied. The deduction function then infers those literals that hold as a consequence of the extended truth assignment. The function returns a pair consisting of a set of theory literals entailed by $\hat{Th}(\theta_2, e)$ and a flag res whose value is $\perp$ if $\hat{Th}(\theta_2, e)$ or θ_2 is inconsistent and $\top$ otherwise. The function propagate calls itself recursively until no further propagation is possible. After deduction returns, if f is not yet satisfied then a further truth assignment is made and DPLL(T) calls itself recursively.

The key difference between the lazy-basic approach and the DPLL(T) approach is that where the lazy-basic approach computes a complete satisfying assignment to the variables of the propositional skeleton before investigating the satisfiability of the corresponding theory formula, the DPLL(T) approach incrementally investigates the consistency of the posted constraints as propositional variables are assigned. Further, it identifies literals, ℓ, such that $\hat{Th}(\theta, e) \models \ell$, allowing $e(\ell)$ to be assigned during propagation. It is the interplay between propositional satisfiability, posting constraints and the consistency of the store $\hat{Th}(\theta, e)$ that is at the heart of this investigation.

4. Propagation and Reification

This section provides a framework for incorporating theory propagation into the propagation framework of the SAT solver from [25]. The approach is based on reifying theory literals with logical variables. As will be illustrated in subsequent sections, this allows the use of the control provided by delay declarations to realise theory propagation. The integration is almost seamless since the base SAT solver is also realised using logical variables and by exploiting the control provided by delay declarations.

4.1. Theory propagation

There are three major steps in setting up a DPLL(T) solver for some problem ϕ: setting up the encoder map e, linking each theory literal in a problem with a logical variable; posting theory propagators (adding constraints) that reify the theory literals with the logical variables provided by e; posting the SAT problem defined by the propositional skeleton $e(\phi)$, then solving this problem. The code in Figure 2 describes the high level call to the solver.

Set up. Where `Prob` is an SMT formula over some theory, let lit(`Prob`) be the set of literals occurring in `Prob`. `TheoryLiteral` is a list of pairs $\ell - e(\ell)$ (or rather, $\ell \leftrightarrow e(\ell)$), where $\ell \in lit$(`Prob`), that defines the encoder mapping e. `Skeleton` represents the propositional skeleton of the problem, e(`Prob`). `Vars` represents the set of variables $e(\ell)$, where $\ell \in lit$(`Prob`). The role of the predicate `setup(+,-,-,-)` is, given `Prob`, to instantiate the remaining variables.

Theory propagators. The role of `post_theory` is to set up predicates to reify each theory literal. The control on these predicates is key; the predicates need to be blocked until either $e(\ell)$ is assigned, or the literal (or its negation) is entailed by the constraint store $\hat{Th}(\theta, e)$. That is, the predicate for $\ell - e(\ell)$ will propagate in one of four ways:

- If $\hat{Th}(\theta, e) \models \ell$ then $e(\ell) \mapsto true$
- If $\hat{Th}(\theta, e) \models \neg\ell$ then $e(\ell) \mapsto false$
- If $e(\ell) = true$ then the store is updated to $\hat{Th}(\theta \cup \{e(\ell) \mapsto true\}, e)$
- If $e(\ell) = false$ then the store is updated to $\hat{Th}(\theta \cup \{e(\ell) \mapsto false\}, e)$

```
dpll_t(Prob):-
    setup(Prob, TheoryLiterals, Skeleton, Vars),
    post_theory(TheoryLiterals),
    post_boolean(Skeleton),
    elim_vars(Vars).
```

Figure 2: Interface to the DPLL(T) solver

Boolean propagators. The role of `post_boolean` is to set up propagators for the SAT part of the problem e(`Prob`). This is a call to `problem_setup` as described in [25]. Search is then driven by assignments to the variables using `elim_vars`.

Implementing the interface provided by predicates `setup` and `post_theory`, together with the SAT solver from [25] results in a DPLL(T) SMT solver. Note that the propagators posted for the theory and Boolean components are intended to capture the spirit of the function propagate from Figure 1. Indeed, the integration between theory and Boolean propagation is even tighter than the algorithm indicates. Rather than performing unit propagation to completion, then performing theory propagation, then repeating, here the assignment of a Boolean variable is immediately communicated to the theory. This tactic is known as immediate propagation and is a natural consequence of using Prolog's control to implement propagators. Immediate propagation does away with the need to analyse failure to determine an unsatisfiable core when a set of theory constraints is unsatisfiable, but attracts a cost in monitoring the entailment status of the theory literals.

4.2. Labelling strategies

The solvers presented in [25] maintain Boolean variables in a list and `elim_vars` assigns them values in the order in which they occur; the list has typically been ordered by the number of occurrences of the variables in the SAT instance before the search begins, the most frequently occurring assigned first. This tactic is straightforward to accommodate into a solver coded in Prolog. The desire for improved performance motivates the adoption of more sophisticated heuristics for variable assignment. Although orthogonal to the theme of theory propagation, the description of the SMT solver would be incomplete without an explanation of labelling.

One classic strategy for labelling that is also straightforward to incorpo-

rate into a solver written in a declarative language is to rank variables by their number of occurrences in clauses of minimal size [29]. This associates a weight to each unbound variable according to its number of occurrences in the unsatisfied clauses of the (Boolean) problem. The ranking weights variables with fewer unbound literals less heavily than those in clauses with a greater number of unbound literals. A variable with greatest weight is selected for labelling, the aim being to assign one that is more likely to lead to propagation.

A refinement of this idea is to apply lookahead [36] in conjunction with this labelling tactic. Each variable with greatest weight, and therefore each candidate for labelling, is speculatively assigned a truth value. For example, if `X` is assigned *true* and this results in failure, then in order to satisfy the propositional formula (skeleton) then `X` must be assigned $false$. Likewise, if failure occurs when `X` is assigned $false$ then `X` must be *true*. Moreover, if one variable can be assigned using lookahead, then often so can others, hence this tactic is repeatedly applied until no further variables can be bound. Thus lookahead is tried before any variable is assigned by search.

Scoping this activity over the variables of greatest weight limits the overhead of lookahead. The net effect is to direct the search away from variable assignments that will ultimately fail. Lookahead can be considered to be dual of clause learning since the former seeks to avoid inconsistency by considering assignments that are still to be made, whereas the latter diagnoses an inconsistency from an assignment that has previously been made. The case for lookahead versus learning has been studied [36], but in a declarative context, particularly one where backtracking is supported, lookahead is very simple to implement, requiring less than 20 lines of additional code in the SAT solver. Clause learning could be added the solver following the techniques discussed in [25], though it does not fit elegantly with chronological backtracking since one cannot straightforwardly add a clause in one branch of the search which is subsequently applied in another branch.

4.3. Calculating an unsatisfiable core

Given an unsatisfiable SMT problem, it can be useful to find an unsatisfiable core of this problem, that is, a subset of the theory literals, $\Sigma' \subseteq \Sigma$, such that $\hat{Th}_{\Sigma'}(\theta, e)$ is not satisfiable for any assignment θ, and for all $\Sigma'' \subset \Sigma'$ there exists a θ such that $\hat{Th}_{\Sigma''}(\theta, e)$ is satisfiable.

The unsatisfiable core needs to be calculated in the lazy-basic approach (in [25] an algorithm adapted from [28] was used). Further, in the application

```
(1)  function findcore (e = [t_1 ↦ x_1, ... , t_n ↦ x_n] : Σ → X,
(2)                     f : CNF formula, c : int, core : Σ → X)
(3)  begin
(4)      if (e = [ ])
(5)          return core;
(6)      else if (c = 0)
(7)          core' := [t_1 ↦ x_1, t_n ↦ x_n] ∪ core;
(8)          return findcore([t_2 ↦ x_2, ... , t_{n-1} ↦ x_{n-1}], f, ⌊(n-1)/2⌋, core');
(9)      else
(10)         i := 1; j := n;
(11)         if (¬DPLL(T)(f, ∅, [t_{c+1} ↦ x_{c+1}, ... , t_n ↦ x_n] ∪ core))
(12)             i := c + 1;
(13)         endif
(14)         if (¬DPLL(T)(f, ∅, [t_i ↦ x_i, ... , t_{n-c} ↦ x_{n-c}] ∪ core))
(15)             j := n - c;
(16)         endif
(17)         if (c = 1)
(18)             c' := 0;
(19)         else
(20)             c' := ⌊(c+1)/2⌋;
(21)         endif
(22)         return findcore([t_i ↦ x_i, ... , t_j ↦ x_j], f, c', core);
(23)     endif
(24) end
```

Figure 3: Finding an unsatisfiable core

to type recovery problems, it is useful to be able to diagnose the cause of unsatisfiability. An unsatisfiable core for the type recovery problems is typically small and this motivates an algorithm that attempts to aggressively prune out literals that are not in a core. Such an algorithm is presented in Figure 3.

The first argument to findcore is (an ordered representation of) a partial encoder mapping from theory literals to propositional variables; the second argument is a propositional formula, namely $e(\phi)$, the propositional skeleton of the initial problem; the third argument is an integer, giving the number of elements of the mapping on literals that will be pruned from one end (and

then the other end) in order to investigate satisfiability; the fourth argument is a partial mapping from theory literals to propositional variables, where the theory literals are part of the unsatisfiable core. The initial call to the function is findcore(e,$e(\phi)$,$\lceil \frac{m}{2} \rceil$,$\emptyset$), where e is the complete encoder map for Σ, $[t_1 \mapsto e(t_1), \ldots, t_m \mapsto e(t_m)]$.

The algorithm removes c elements from the beginning of the mapping (represented as a list) and tests the resulting problem for satisfiability. If the problem remains unsatisfiable, the c elements removed are not part of the unsatisfiable core and can be pruned all at once. This is repeated for the end of the mapping. The c value begins large and is logarithmically reduced until it has value 0, at which point the first and last elements of the list representing the mapping must be in the core. The function findcore is then again recursively called with these end points removed and the process continues until a core has been found.

Our findcore algorithm is related to the QuickXplain algorithm of [30] which likewise computes an unsatisfiable core by removing blocks of consistent constraints. QuickXplain recursively divides a set of constaints into two subsets. If the first subset is inconsistent, then the second can be discarded immediately in the search for a core. Otherwise, some constraints from the second are merged with constraints from the first to derive a core. This divide-and-conquer algorithm resembles a precedessor of findcore though, crucially, the above incarnation of the algorithm attempts to remove the first $c_1 = \lceil \frac{m}{2} \rceil$ constraints, then the last c_1, then the first $c_2 = \lceil \frac{c_1}{2} \rceil$, then the last c_2, etc, as it converges onto a core. This appears to be a more aggressive pruning strategy than that applied in QuickXplain which maintains the first subset intact (see [30, Figure 1, Line 9]) whilst pruning the second.

The following proposition asserts that the algorithm correctly computes a core.

Proposition 1. If $\hat{Th}_\Sigma(\theta, e)$ is unsatisfiable for any assignment θ and e is an encoder mapping for Σ, then function findcore returns an encoder mapping that represents an unsatisfiable core, $\Sigma' \subseteq \Sigma$.

Proof. First it is argued that findcore terminates. Where a call to findcore is findcore($e, f, c, core$) consider the ordered pair $\langle |e|, c \rangle$. For each recursive call to findcore the value of this pair lexicographically decreases, therefore by induction findcore terminates.

Next it is demonstrated that findcore returns an encoder mapping representing an unsatisfiable set of variables. A precondition of a call to findcore

is that $\hat{Th}_\Sigma(\theta, e)$ is unsatisfiable. Since *core* is initially empty, the encoder $e \cup core$ represents an unsatisfiable set of variables. Lines (12) and (15) are the two places where findcore changes $e \cup core$. Observe that the conditions on line (11) and (14) state that the change of e to e' (where e' is the updated encoder mapping) occurs only when $e' \cup core$ represents an unsatisfiable set of variables. Hence that $e \cup core$ represents an unsatisfiable set of variables is invariant through the algorithm. findcore returns when $|e| = 0$, therefore returning $core = e \cup core$ which represents an unsatisfiable set of variables.

Lastly it is demonstrated that the set of variables represented by the encoder mapping returned represents an unsatisfiable core. Note that if some encoder mapping d represents an unsatisfiable set, then $d' \supseteq d$ also represents an unsatisfiable set. Variable mappings are added to *core* on line (7). Suppose $t \mapsto x$ is added to *core* at line (7). In the preceding call with $c = 1$, either $(e \setminus [t_1 \mapsto x_1]) \cup core$ represents a satisfiable set or it does not. (The argument for $t_n \mapsto x_n$ is symmetric.) If satisfiable, then $t_1 \mapsto x_1$ is required for unsatisfiability, and this is the $t \mapsto x$ added in line (7). If unsatisfiable, then the $t \mapsto x$ added in line (7) is $t_2 \mapsto x_2$.

Note that in the initial call to findcore with mapping e (where $|e| = m$) all elements of e to the left of $t_2 \mapsto x_2$ are not part of the representation of the unsatisfiable core since they are removed by lines (11)-(13). The number of elements removed is described by Lemma 1, that is, $m-1$. Therefore $t_2 \mapsto x_2$ is the only element remaining and this must be part of the representation of the unsatisfiable core, since otherwise it would have been removed by lines (14)-(16). Therefore no redundant elements are added and findcore returns an unsatisfiable core. □

Lemma 1. Where $m \in \mathbb{N}$, let $c_1(m) = \lceil \frac{m}{2} \rceil$ and $c_i(m) = \lceil \frac{c_{i-1}(m)}{2} \rceil$, if $c_{i-1}(m) > 1$ and $c_i(m) = 0$ otherwise. Then $\sum_i c_i(m) \geq m - 1$.

Proof. The proof proceeds by induction. Suppose $m = 1$, then $c_1(m) = 1 \geq m - 1$. Suppose $m = 2$, then $c_1(m) = 1$ and $c_2(m) = 0$, hence $\sum_i c_i(m) = 1 \geq m - 1$. Now suppose m is odd, that is $m = 2k - 1$ (where $k \in \mathbb{N}$, $k \geq 1$), then $c_1(2k-1) = \lceil \frac{2k-1}{2} \rceil = k$ and $c_2(2k-1) = c_1(k)$. Hence $\sum_i c_i(2k-1) = k + \sum_i c_i(k) \geq 2k - 1 \geq 2k - 2$. Suppose m is even, that is $m = 2k$ (where $k \geq 1$) then $c_1(2k) = \lceil \frac{2k}{2} \rceil = k$ and $c_2(2k) = c_1(k)$. Hence $\sum_i c_i(2k) = k + \sum_i c_i(k) \geq 2k - 1$. □

Note that when this lemma is applied, $c_i(m)$ corresponds to the third argument of findcore (c) at iteration i. This value is a function of m, the size of

the initial encoder map.

5. Instantiation for Rational-Trees

The theory component of an SMT solver requires a decision procedure for determining the satisfiability of a conjunction of theory literals. Unification is at the heart of Prolog and many Prolog systems are based on rational-tree unification, hence a decision procedure for conjunctions of rational-tree constraints comes essentially for free. This can be coupled with the control provided by delay declarations to reify rational-tree constraints, hence implementing the interface described in section 4. The code in Figure 4 demonstrates the use of delay to realise theory propagation over rational-tree constraints via reification.

An SMT problem over rational-trees consists of Boolean combinations of theory literals ℓ. The call to `setup/4` will instantiate `TheoryLiterals` to a list of pairs of the form $\ell - e(\ell)$; the propositional skeleton and a list of the $e(\ell)$ variables are also produced. In the following, a labelled literal `eqn(Term1, Term2)-X` is discussed. `post_theory` sets up propagators for each theory literal in two steps. `theory_wait` propagates from the theory constraints into the Boolean variables.

`theory_wait` uses the builtin control predicate `when/2`, which blocks the goal in its second argument until the first argument evaluates to *true*. In this instance the condition `?=(Term1, Term2)` is *true* either if `Term1` and `Term2` are identical, or if the terms cannot be unified. That is, if `Term1=Term2` is entailed by the store then `theory_prop` is called and assigns `X=true`. Similarly, if the constraint is not consistent with the store, then `Term1` and `Term2` cannot be unified and again `theory_prop` reflects this by assigning `X=false`. In the opposite direction, `bool_wait` communicates assignments made to Boolean variables to the theory literals. The predicate is blocked on the instantiation of the logical variables, waking when they become `true` or `false`. When `true` the constraint must hold so `Term1` and `Term2` are unified. When `false`, it is not possible for the two terms to be unified, hence the constraint is discarded and the call to `bool_wait` succeeds. Note that it is not possible to post a constraint that asserts that two terms cannot be unified, since the control predicate `dif/2` is defined as:

```
dif(X, Y) :- when(?=(X, Y), X \== Y).
```

```
post_theory([]).
post_theory([eqn(Term1,Term2)-X|Rest]) :-
    setup_reify(X, Term1, Term2),
    post_theory(Rest).

setup_reify(X, Term1, Term2) :-
    bool_wait(X, Term1, Term2),
    theory_wait(X, Term1, Term2).

:- block bool_wait(-, ?, ?).
bool_wait(true, Term1, Term2) :-
    Term1 = Term2, !.
bool_wait(false, _Term1, _Term2).

theory_wait(X, Term1, Term2) :-
    when(?=(Term1, Term2), theory_prop(X, Term1, Term2)).

theory_prop(X, Term1, Term2) :-
    Term1 == Term2 ->
        X = true
    ;
        X = false
    .
```

Figure 4: Theory propagation for rational-tree constraints

That is, it blocks until either `X` and `Y` are identical or they cannot be unified, then tests whether or not they are identical. Hence `dif/2` acts as a test, rather than a propagating constraint. Consistency of the store is maintained by `theory_wait`; if `X=false` and the constraint is discarded, then later it is determined that `Term1=Term2`, `theory_wait` will attempt to unify `X` with `true`, which will fail. Finally, `post_boolean` sets up the propositional skeleton for the solver from [24].

6. Instantiation for Linear Real Arithmetic

Many Prolog systems come with the CLP($\mathcal{R}$) constraints package, which can determine consistency of conjunctions of linear arithmetic constraints.

This decision procedure makes quantifier-free linear real arithmetic a sensible theory for the solver. The challenge is to implement reification for the constraints, an operation not directly supported.

The code in Figure 5 demonstrates the integration of linear real arithmetic as realised by CLP($\mathcal{R}$) into the DPLL(T) scheme. It assumes that the input problem has been normalised so that all the constraint predicates are drawn from $=$, $=<$ and $<$. The propagators, `theory_wait`, are blocked on two variables. The first of these is the labelling variable e(`C`) – if this is instantiated, the appropriate constraint is posted. To complete the reification, the propagators need to detect the entailment of the linear constraint (or its negation). This can be achieved using the builtin `entailed/1`, however the control for ensuring that this is called at an appopriate time is less obvious.

Once a new constraint has been posted (or once the constraint store has changed) other constraints or their negations might be entailed and this needs to be detected and propagated. The communication between the propagators to capture this is achieved with the second argument to `theory_wait`. Each propagator is set with its second argument the same logical variable (`Y` in the code) and the propagators are blocked on this second argument. When a constraint is posted, `Y` is instantiated, `Y = prop(_)`. This wakes all active propagators which either propagate or block again on the new variable. An alternative approach, which would invoke the propagators less frequently, would be to only wake up the activate propagators for those constraints that share a variable with the posted constraint.

It should be emphasised, however, that although a linear solver is interesting for self-contained Prolog applications, this theory is supported by a number of off-the-shelf SMT solvers; the approach presented in this paper is primarily designed for constraint theories that are unavailable in standard SMT distributions.

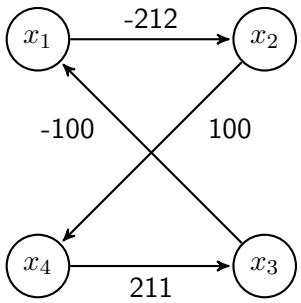
7. Instantiation for Difference Logic

Thus far it has been demonstrated how theory propagation can be realised for SMT solvers over rational-tree unification and linear constraints, both of which are constraint systems that are built-in to Prolog. This begs the question of whether logical variables can be used to orchestrate theory propagation for a solver over a theory, such as difference constraints, that is not readily available in Prolog. The challenge is to find a way for efficiently

```
post_theory(TheoryLiterals):-
    setup_reify(TheoryLiterals, _).

setup_reify([], _).
setup_reify([C-V|Cs], Y) :-
    negate(C, NegC),
    theory_wait(V, Y, C, NegC),
    setup_reify(Cs, Y).

negate(X =< Y, X > Y).
negate(X < Y, X >= Y).
negate(X = Y, X =\= Y).

next_var(Y, Z) :-
    var(Y), !, Y = Z.
next_var(prop(Y), Z) :-
    next_var(Y, Z).

:- block theory_wait(-, -, ?, ?).
theory_wait(V, Y, C, _NegC) :-
    V == true, !,
    {C}, Y = prop(_).
theory_wait(V, Y, _C, NegC) :-
    V == false, !,
    {NegC}, Y = prop(_).
theory_wait(V, Y, C, _NegC) :-
    nonvar(Y), entailed(C), !,
    V = true.
theory_wait(V, Y, _C, NegC) :-
    nonvar(Y), entailed(NegC), !,
    V = false.
theory_wait(V, Y, C, NegC) :-
    next_var(Y, U),
    theory_wait(V, U, C, NegC).
```

Figure 5: Theory propagation for linear real arithmetic

Figure 6: Illustrating Floyd-Warshall



$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} 0 & \infty & -100 & \infty \\ -212 & 0 & \infty & \infty \\ \infty & \infty & 0 & 211 \\ \infty & 100 & \infty & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} -1 & 211 & -101 & 110 \\ -213 & -1 & -313 & -102 \\ 99 & 311 & -1 & 210 \\ -113 & 99 & -213 & -2 \end{bmatrix} \end{array}$$

deciding which theory constraints are entailed or disentailed, and then communicating this information to the SAT solver.

Difference constraints are a strict subclass of linear constraints in which each constraint has unary coefficients and is over at most two variables. To be precise, each constraint must take the form $x_i - x_j \leq c$, where x_i and x_j are variables, and c is a constant. A designated variable x_0 is interpreted as zero to encode unary constraints, and the constants themselves are either integers, rationals or reals. Difference constraints are emblematic of other two variable systems and can be readily modified to richer domains such as Octagons [41]. For this study we focus on integer difference logic. Integer difference logic (QF_IDL) is an SMT problem where integer difference constraints are composed with logical connectives.

7.1. The Floyd-Warshall algorithm

Decision procedures for systems of difference constraints are often obtained by viewing a system as a weighted directed graph. To illustrate, consider the constraints $(x_1 - x_2 \leq -212) \wedge (x_3 - x_1 \leq -100) \wedge (x_4 - x_3 \leq 211) \wedge (x_2 - x_4 \leq 100)$, which can be interpreted as the graph given in the left column of Figure 6. The graph in turn can be represented by the adjacency matrix given in the middle column. Solving the all pairs shortest path problem then populates the matrix with entries that describe the shortest paths between any two variables, as shown in the right column of Figure 6. Observe that the diagonals of this final matrix are negative, indicating that the graph contains negative cycles, showing that the system of constraints is inconsistent.

The Floyd-Warshall algorithm [18, 55], which is in $\mathcal{O}(n^3)$, solves the all pairs shortest path problem where n is the number of variables. Moreover,

Figure 7: Floyd-Warshall algorithm: non-incremental and incremental versions

```
function
floyd-warshall(m, n)
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            m_{i,j} := min(m_{i,j}, m_{i,k} + m_{k,j})
        endfor
        if (m_{i,i} < 0)
            return UNSAT
        endif
    endfor
endfor
return SAT
```

```
function
floyd-warshall(m, n, x_p − x_q ≤ c)
for i = 1 to n
    for j = 1 to n
        m_{i,j} := min(m_{i,j}, m_{i,p} + c + m_{q,j})
    endfor
    if (m_{i,i} < 0)
        return UNSAT
    endif
endfor
return SAT
```

an incremental version can be formulated that is only $\mathcal{O}(n^2)$ for each new added constraint [5]. Both versions are shown below in Figure 7. The non-incremental version compares all possible paths through the graph between each pair of variables x_i and x_j by checking whether the path between them can be shortened by passing through another variable x_k.

The incremental version takes a new constraint $x_p - x_q \leq c$, and checks whether the path between x_i and x_j could be reduced by travelling via the new edge that represents the constraint. That is, if the cost of travelling from x_i to x_p, then from x_p to x_q (a distance of c) and then from x_q to x_j, is less than that of moving from x_i to x_j directly. Observe that $m_{p,q}$ will be updated to c if $c < m_{p,q}$ when $i = p$ and $j = q$ since $m_{i,i} = 0$ and $m_{j,j} = 0$, assuming consistency. In both cases the consistency check is placed inside the main loop so that negative cycles cause rapid failure (though failure could be made faster at the expense of decomposing the innermost loop).

Figure 8: Setting up the Floyd-Warshall matrix and the watch matrix

```
post_theory(Encoding, Store) :-
    build_store(Encoding, Store),
    setup_reify(Encoding, Queue),
    process_queue(Queue, Store).

build_store([], Store) :-
    empty_avl(Matrix), empty_avl(Watch),
    Store = store(Matrix, 0, Watch).
build_store([(X-Y =< C)-Prop | Rest], Store) :-
    build_store(Rest, StoreRest),
    StoreRest = store(Matrix1, N1, Watch1),
    Store = store(Matrix3, N3, Watch3),
    add_var(X, N1, Matrix1, N2, Matrix2),
    add_var(Y, N2, Matrix2, N3, Matrix3),
    (avl_fetch(X-Y, Watch1, EntL-DisL) ->
        avl_store(X-Y, Watch1, [C-Prop | EntL]-DisL, Watch2)
    ;
        avl_store(X-Y, Watch1, [C-Prop]-[], Watch2)
    ),
    NegC is -(C + 1),
    (avl_fetch(Y-X, Watch2, EntL2-DisL2) ->
        avl_store(Y-X, Watch2, EntL2-[NegC-Prop | DisL2], Watch3)
    ;
        avl_store(Y-X, Watch2, []-[NegC-Prop], Watch3)
    ).
```

7.2. Theory propagation in difference logic

For rational-trees, a built-in test can be used within a wait declaration to block a goal until a rational-tree constraint is entailed or disentailed, which binds the propositional variable that reifies the constraint when the goal resumes. It is less obvious how to program an analogous control structure for difference logic in order to realise theory propagation. The rest of this section explains how this control can be achieved.

Figure 9: Propagating from the SAT solver to the theory solver

```
setup_reify([], _).
setup_reify([(X-Y =< C)-Prop | Rest], Queue) :-
    bool_wait(Prop, X, Y, C, Queue),
    setup_reify(Rest, Queue).

:- block bool_wait(-, ?, ?, ?, ?).
bool_wait(Prop, X, Y, C, Queue) :-
    Prop == true, !,
    insert_queue(Queue, X, Y, C).
bool_wait(Prop, X, Y, C, Queue) :-
    Prop == false, !,
    NegC is -(C + 1),
    insert_queue(Queue, Y, X, NegC).

insert_queue(Queue, X, Y, C) :-
    var(Queue), !,
    Queue = [(X-Y =< C) | _Cons].
insert_queue([_Con | Cons], X, Y, C) :-
    insert_queue(Cons, X, Y, C).

:- block process_queue(-, ?).
process_queue(Queue, Store1) :-
    nonvar(Queue), !,
    Queue = [(X-Y =< C) | Cons],
    process_constraint(X-Y =< C, Store1, Store2),
    process_queue(Cons, Store2).
```

7.2.1. Data-structures

The proposal is to shadow the Floyd-Warshall matrix with a square matrix of the same dimension that records which constraints in the SMT formula are possibly entailed or disentailed. This matrix, dubbed the watch matrix, is so named because it is inspired by watch literals [42] that are key to efficient SAT solving. The matrix is a control structure for efficiently identifying which propositional variables should be bound, and to what truth values,

when an entry in the Floyd-Warshall matrix is updated.

The watch matrix is constructed once. This matrix, in conjunction with the Floyd-Warshall matrix, constitutes the store. The Floyd-Warshall matrix maps each variable pair x-y (its indices) to a value $v \in \mathbb{Z} \cup \{\infty\}$ which is the length of the shortest known path from x to y (∞ used to indicate the absence of any such path). An entry of v can be interpreted as asserting the inequality $x - y \leq v$, which vacuously holds if $v = \infty$.

The watch matrix maps the indices x-y to a pair `EntL-DisL` where `EntL` and `DisL` are themselves lists of pairs, referred to as the entailed pairs and the disentailed pairs. Each of the entailed pairs takes the form $c - \texttt{Prop}$ where `Prop` is a propositional variable that reifies a constraint $x - y \leq c$ which occurs somewhere in the SMT formula. When the $x - y$ entry is updated with v in the Floyd-Warshall matrix, the corresponding list `EntL` is traversed to find those pairs $c - \texttt{Prop}$ for which $v \leq c$. Each pair corresponds to a constraint $x - y \leq c$ in the formula that is entailed and moreover reified with `Prop`. Thus `Prop` can be bound to `true` thereby achieving partial theory propagation.

Complete theory propagation is realised by additionally recording in `DisL` those disentailed pairs $(-c - 1) - \texttt{Prop}$ for which there exists a constraint $y - x \leq c$ in the SMT formula. When $v \leq -c - 1$ it follows that $x - y \leq -c - 1$, thus $y - x \leq c$ is disentailed since $y - x \leq c < c + 1 \leq y - x$. Disentailment is communicated to the SAT solver by setting `Prop` to false. Note that `Prop` may already be bound, though not necessarily to the same truth value, in which case inconsistency is detected. Detecting all disentailed constraints again only involves a list traversal and low-cost inequality checks.

7.2.2. Setup

The watch matrix is set up in tandem with the Floyd-Warshall matrix by `build_store` predicate. This predicate traverses the encoder map, considering each reified constraint `(X-Y =< C)-Prop` in turn. Both matrices are represented by AVL trees so that elements of these matrices are accessed and updated by `avl_fetch` and `avl_store` respectively. The body of `build_store` adds `C-Prop` to the list of entailed pairs and `NegC-Prop` to the list of disentailed pairs where $\texttt{NegC} = -\texttt{C} - 1$. The calls to the `add_var` predicate, when necessary, add a new row and column to the Floyd-Warshall matrix and populates the new diagonal element with zero and any other new entries with ∞.

Figure 10: Propagating from the theory solver to the SAT solver

```
process_constraint(X-Y =< C, StoreIn, StoreOut) :-
    StoreIn = store(Matrix1, N, Watch),
    StoreOut = store(Matrix3, N, Watch),
    avl_fetch(X-Y, Matrix1, C_XY),
    min(C_XY, C, Min),
    (C == Min ->
        matrix_update(X-Y, C, Matrix1, Matrix2, Watch),
        floyd_warshall(N, Matrix2, Matrix3, Watch)
    ;
        Matrix3 = Matrix1
    ).

matrix_update(Key, Value, Matrix1, Matrix2, Watch) :-
    (avl_fetch(Key, Watch, EntL-DisL) ->
        true
    ;
        EntL = [], DisL = []
    ),
    entailed(EntL, Value),
    disentailed(DisL, Value),
    avl_store(Key, Matrix1, Value, Matrix2).

entailed([], _).
entailed([C-Prop | Rest], Min) :-
    (Min =< C -> Prop = true ; true),
    entailed(Rest, Min).
```

7.2.3. Posting difference constraints

Reification provides a channel for passing information from the theory solver to the SAT solver. Conversely, when the SAT solver binds a propositional variable that reifies a difference constraint, either the constraint, or its negation, must be posted (added) to the store. As a consequence of the update, incremental Floyd-Warshall should be applied together with any ensuing theory propagation.

Logical variables also provide a mechanism for coordinating these events, which must be fully backtrackable. This can be elegantly achieved with an open list that queues up the difference constraints that are to be posted to the store. The predicate `insert_queue` inserts a difference constraint at the end of `Queue`. The predicate `process_queue` blocks until a constraint is in the queue at which point the constraint is passed onto `process_constraint` that activates Floyd-Warshall, before inspecting, and if necessary blocking, until another element appears in the `Queue`. Observe how the store is updated as the constraints are processed. The predicate `process_constraint` invokes Floyd-Warshall, though only if the new constraint $x - y \leq c$ has a constant c that is strictly smaller than the value stored in the matrix at index $x - y$. The update is performed by `matrix_update` which extracts two lists from the watch matrix: the entailed pairs and the disentailed pairs. The predicate `entailed` serves to illustrate how lightweight this form of theory propagation actually is once the watch matrix has been constructed. The predicate `disentailed` is defined analogously.

The predicate `bool_wait` which, recall, waits until a reification variable is set to a truth-value, in this setting merely pushes the constraint, or its negation, into the queue.

8. Experimental Results

8.1. Rational-tree solver

The DPLL(T) solver for rational-trees has been coded in SICStus Prolog 4.2.1, as described in section 5. Henceforth this will be called the Prolog solver. To assess this solver it has been applied to a benchmark suite of 84 type recovery problems, its target application. The first eight benchmarks are drawn from compilations at different optimisation levels of three small programs manufactured to check their types against those derived by the solver. These benchmarks are designed to check that the inferred types match against those prescribed in the source file, and also assess the robustness of the type recovery in the face of various compilation modes. The remaining benchmarks are taken from version 8.9 of the coreutils suite of programs, standard UNIX command line utilities such as *wc*, *uniq*, *echo* etc. With an eye to the future, the DynInst toolkit [40] was used to parse the binaries and reconstruct the CFGs. This toolkit can recover the full CFG for many obfuscated, packed and stripped binaries, and even succeeds at determining indirect jump targets. CFG recovery is followed by SSA conversion

which, in turn, is followed by the generation of the type constraints, and the corresponding SMT formula complete with its propositional skeleton. The latter rewriting steps are naturally realised as a set of Prolog rules.

To the best of the authors' knowledge, this work represents the first time that recursive types have been automatically derived, hence it is not possible to compare to previous approaches. Furthermore, no comparison is made with an open source SMT solver equipped with rational-trees since the authors are unaware of any such system. Nevertheless, to provide a comparative evaluation a lazy-basic SMT solver based on an off-the-shelf SAT solver, PicoSAT [6], has been constructed. This solver is also implemented in SICStus Prolog 4.2.1 but uses bindings to PicoSAT to solve the SAT formulae. PicoSAT, though small by comparison with some solvers at approximately 6000 lines of C, applies learning, random restarts, etc, a range of tactics not employed in the Prolog SAT solver. This SMT solver will henceforth be called the hybrid solver. However, crucially, the hybrid solver does not apply theory propagation; it simply alternates SAT solving with satisfiability testing following the lazy-basic approach, which is all one can do when the SAT solver is used as a black box.

The experiments were run on a single core of a MacBook Pro with a 2.4GHz Intel Core 2 Duo processor and 4GB of memory. A selection of the results are given in Table 1. The first column gives the binary from which the constraints were generated, the second column the number of instructions in the binary, the third the number of clauses in the problem, the fourth the number of propositional variables, and the fifth the number of theory variables. In terms of timings, the sixth column records the runtime in seconds to find a model or a core for the Prolog solver, the seventh gives the number of times the Prolog SMT solver was called, the eighth gives the runtime in seconds to find a model or a core for the hybrid solver, and the final column gives the number of times the PicoSAT solver was called. To clarify, consider benchmark 1. The SMT formula is satisfiable, hence a core is not derived, and the problem is solved with just one call to the Prolog SMT solver. The hybrid solver also requires just one call but this, in turn, requires PicoSAT to be invoked 796 times, on all but the last occasion adding a single blocking clause to the propositional skeleton. By way of contrast, benchmark 9 is unsatisfiable hence a core is computed that pinpoints a type conflict. The Prolog SMT solver is invoked 51 times to identify this core; the hybrid SMT solver requires exactly the same number of calls, hence the number is not repeated in the table. However, these 51 calls to the hybrid

solver cumulatively require 536 invocations of PicoSAT. On occasions the hybrid solver terminated with a memory error[2], indicated by seg, invariably after several hours of computation. The fault is repeatable.

In addition to these timing results, the recursive types inferred for merge-sort, as well as those for iterative-sum and recursive-sum, have been checked against the types prescribed in the source. The sum programs both build list of integers but then traverse them in different ways. Another point not revealed from the table is that the largest benchmarks can take over 20 minutes to parse, reconstruct the CFG, perform SSA conversion and then generate the SMT formula. Thus the time required to solve the SMT formulae does not exceed the time required to generate them, at least for the Prolog solver.

8.2. Integer difference logic solver

One of the attractions of the current work is that new theories can be coded in Prolog and be integrated straightforwardly into the SMT solver via reification. For rational-trees a comparison was made against a hybrid solver using PicoSAT as a SAT engine. Since the theory of quantifier-free integer difference logic (QF_IDL) is a standard part of SMT packages, for this theory a more direct comparison between the solver presented and an off-the-shelf solver can be made. That said, off-the-shelf solvers deploy learning, random restarts, among other things, so as to not get lost in the search space, whereas Prolog difference logic solver does not even apply lookahead.

For this more demanding strength test, the Prolog-based SMT solver is benchmarked against the open-source CVC3 and CVC4 solvers, which both consist of many hundreds of thousands of lines of C++ code and have performed well in the SMT competitions [7]. Problems from the latest (2013) version of the SMT-LIB library of SMT benchmarks [4] were used for testing and evaluation. SMT-LIB provides benchmarks for many SMT theories, notably QF_IDL problems. Moreover, benchmarks from this suite are conveniently labelled according to whether they are satisfiable or not, making it straightforward to test the Prolog solver for correctness, even for unsatisfiable instances.

To read the instances, the Prolog solver was extended with a parser for the smtlib2 input language, written using flex and bison, that outputs an

[2]This bug has been fixed in the forthcoming SICStus 4.3, though at the time of writing the latest version available is 4.2.3

abstract syntax tree for an SMT instance represented as a single Prolog term. To resolve overloading on the equality operator, which can be interpreted either as logical bi-implication or as a relational arithmetical operator, the abstract syntax tree was traversed to infer types and thereby disambiguate the usage of equality terms.

The benchmarks include both industrial problems, and difficult crafted problems designed specifically to test the performance of a solver. Instances in the QF_IDL class were ranked according to size, and directed at the Prolog solver and at CVC3 and CVC4. CVC3 was timed using the unix *time* command, measuring overall runtime, while the runtime of CVC4 was measured using its stats command line option. The runtime of the Prolog solver was found using the statistics predicate, though it was only able to resolve the run time with a granularity of ten milliseconds. Table 2 gives a selection of the results taken from the first two hundred benchmarks, which have been pruned to remove any whose run time was less than fifty milliseconds for all three solvers (which removed 35 benchmarks before the table even started). Benchmarks with similar names and performance were also removed to make space for a wider range of instances.

9. Discussion

9.1. Rational-tree solver

The results in Table 1 demonstrate that an SMT solver equipped with an appropriate theory can be used to successful automate the recovery of recursive types, a problem not previously solved.

On no occasion is the hybrid solver faster than the Prolog solver, which suggests that a succinct implementation of theory propagation is more powerful than deploying an off-the-shelf SAT solver as a black box in combination with a handcrafted theory solver using the lazy-basic approach.

It can be observed in Table 1 that many of the problems are unsatisfiable. For these problems an explanation for a type conflict is returned rather than a satisfying type assignment. As a strength test of the solver these problems are good since the exhaustive search required to demonstrate unsatisfiability is more demanding than search for a first satisfying assignment. There are two results that require discussion. Benchmark 4 has an unsatisfiable core of 26 constraints, whereas most cores have less than 10 constraints. This explains why it is relatively slow. Benchmark 7 has timed out, a reminder that large SMT problems can be hard to solve.

Table 1: Benchmarking for a selection of type recovery problems

benchmark	insns	clauses	vars prop	vars theory	SMT time	SMT calls	SAT time	SAT calls
1 iter-sum.O1	296	2047	564	779	14.57	SAT	413.36	796
2 iter-sum.O2	312	2132	586	812	52.34	SAT	seg	
3 recu-sum.O1	302	2129	588	809	15.37	SAT	6382.50	998
4 mergesort.O0	480	3216	888	1220	585.89	70	seg	
5 mergesort.O1	387	2636	718	1011	20.05	SAT	1176.58	1720
6 mergesort.O2	395	2628	713	1017	20.30	SAT	805.93	860
7 mergesort.Os	444	3275	907	1244	>14400		seg	
8 mergesort.O3	2586	15696	3741	6670	1551.23	31	>14400	
9 false	3747	27645	5357	12957	19.46	51	3250.05	536
10 true	3747	27645	5357	12955	19.27	51	3247.02	536
11 tty	3825	28255	5417	13373	20.02	51	3509.06	552
12 sync	3901	28706	5571	13466	70.76	52	3607.01	553
15 hostid	3912	28973	5576	13634	62.70	52	3651.77	550
19 basename	4114	30125	5829	14212	69.48	53	3939.21	544
20 env	4016	29670	5589	13956	22.69	53	3914.54	544
22 uname	4074	31048	5653	15034	32.28	52	3676.94	534
23 cksum	4259	31973	5975	15370	101.85	52	4516.21	554
24 sleep	4442	32993	6343	15637	84.89	51	4876.85	566
29 echo	4310	33087	6064	15571	41.41	51	4723.52	564
30 nice	4397	33057	6000	15719	11.23	51	4907.31	581
33 nl	5719	43834	7692	21240	17.20	56	seg	
34 comm	5563	45401	7790	22797	108.09	53	10667.25	650
42 wc	6377	52105	8818	26713	93.91	52	12681.63	575
43 uniq	6595	52779	9013	27190	35.46	53	13281.49	581
51 join	7946	67168	10844	34688	85.93	60	>14400	
53 sha384sum	11612	78776	16419	36153	191.87	53	>14400	
54 cut	8173	68332	11248	36736	185.84	60	>14400	
58 ln	9369	83877	12668	44935	292.21	54	>14400	
61 getlimits	10797	92504	14856	47845	396.81	54	>14000	
66 timeout	12063	98544	16306	50019	126.79	53	>14000	
78 ptx	15919	141197	21850	76881	702.67	55	>14000	
89 mbslen	25895	257132	35148	148102	1935.12	56	>14400	

Note that the time required for type recovery is sensitive to optimisation level, though it is not obvious why different optimisation levels impact on the difficulty of the SMT instance, apart from the obvious effect on code size.

For the unsatisfiable problems, a core of unsatisfiable constraints is calculated using multiple calls to the DPLL(T) solver as indicated. This core can be used to diagnose unsatisfiability, in turn allowing the analysis to be refined to return meaningful information despite the initial result. In the benchmarks unsatisfiability is typically owing to **nop** instructions such as **nop** [rax+rax+0x0]. This instruction does nothing, but has been generated by the compiler with an encoded operand in order to make it a specific size for optimal performance. The indirect addressing is broken down and constraints generated as follows:

$$
\begin{array}{ll}
\textbf{mov}\ \mathrm{A}_1, \mathrm{rax}_1 & T_{A_1} = T_{rax_1} \\
\textbf{add}\ \mathrm{A}_2, \mathrm{rax}_1 & \left(\begin{array}{c} T_{A_2} = \mathrm{basic}(_,\mathrm{int},4) \wedge T_{A_1} = T_{A_2} \wedge \\ T_{rax_1} = T_{A_2} \end{array} \right) \vee \\
 & \left(\begin{array}{c} T_{A_2} = \mathrm{ptr}(\langle [...], \alpha_1, [...] \rangle) \wedge \\ T_{A_1} = \mathrm{ptr}(\langle [...], \alpha_2, [...] \rangle) \wedge \\ T_{rax_1} = \mathrm{basic}(_,\mathrm{int},4) \end{array} \right) \vee \\
 & \left(\begin{array}{c} T_{A_2} = \mathrm{ptr}(\langle [...], \alpha_3, [...] \rangle) \wedge \\ T_{A_1} = \mathrm{basic}(_,\mathrm{int},4) \wedge \\ T_{rax_1} = \mathrm{ptr}(\langle [...], \alpha_4, [...] \rangle) \end{array} \right) \\
\textbf{mov}\ \mathrm{A}_3, [\mathrm{A}_2] & T_{A_2} = \mathrm{ptr}(\langle [...], T_{A_3}, [_, _, _, ...] \rangle) \\
\textbf{nop}\ \mathrm{A}_3 &
\end{array}
$$

The final constraint states that $\mathbf{A}_2$ must have pointer type, hence those for the **add** dictate that one of $\mathbf{A}_1$ and $\mathbf{rax}_1$ must be of basic type, and the other a pointer; however, the first constraint says they have the same type, so the system is inconsistent.

Another unexpected source of inconsistency is the hard-coded pointer addresses sometimes found in **mov** instructions. These are often addresses of strings included in the binary, but also include constructor and destructor lists, added by the linker for construction and destruction of objects. For example, the instruction **mov ebx$_1$**, 0x605e38 appears in the cksum binary, and moves the address of a string into **ebx$_1$** resulting in the constraint $T_{ebx_1} =$

basic(_,int,4). Later however, $\mathbf{ebx}_1$ is dereferenced, which implies that it is a pointer, and conflicts with the earlier inference.

Quite apart from the disjunctive nature of constraints, the sheer number of x86 instructions pose an engineering challenge when writing a type recovery tool; indeed the constraint generator module has taken longer to develop than both SMT solvers together. Moreover, as the above two examples illustrate, type conflicts stem from type interactions between different instructions which makes the type conflicts difficult to anticipate. The result produced from the solver is either a successful recovery of types, or a core of inconsistent types, both of which can be achieved sufficiently quickly. Since the core is typically small, it is of great utility in pinpointing omissions in the type generation phase. It seems attractive to augment the solver with a domain specific language for expressing and editing the type constraints so that they can be refined, if necessary, by a user.

9.2. Integer difference logic solver

From the results in Table 2, together with the 35 benchmarks solved in less than 50ms by all solvers, it can be observed that the Prolog QF_IDL solver performs surprisingly well, solving many benchmark instances in times comparable to a well established SMT solver. We suspect that the performance stems partly from our incremental algorithm and partly from the watch matrices which enable lightweight theory propagation, though it is not straightforward to determine the relative importance of these techniques. The performance is particularly pleasing considering that the solver employs neither learning nor lookahead. Morever, this performance has to be balanced against the engineering effort required to implement and integrate the theory in the Prolog solver, which totalled less than six hundred lines of code. The brevity of the code also facilitates easy modification and experimentation, an advantage of any declarative language.

Timeouts occur with several of the benchmarks. The diamonds benchmarks, handcrafted by Ofer Strichman [52], are particularly hard for the Prolog solver, and indeed CVC3 and CVC4 both also timeout on several of these problems. These benchmarks are recognised as challenging problems [2], for which special tactics have been suggested [47], so it is no great surprise that they cause difficulty, particularly as this Prolog solver does not employ any labelling heuristics. The jobshop and DTP benchmarks are also hand-crafted problems designed to stress a solver.

Table 2: Benchmarking for a selection of QF_IDL problems

benchmark	sat	p-vars	t-vars	cvc3	cvc4	Prolog
super_queen6-1.smt2	✗	264	7	64	77	**10**
jobshop4-2-2-2-4-4-11.smt2	✗	112	17	64	**40**	310
toroidal_queen6-1.smt2	✗	288	7	344	310	**30**
super_queen7-1.smt2	✗	354	8	216	368	**40**
queen8-1.smt2	✓	352	9	288	702	**30**
toroidal_queen7-1.smt2	✓	434	8	132	99	**20**
super_queen8-1.smt2	✗	456	9	444	770	**110**
queen9-1.smt2	✓	450	10	908	462	**120**
toroidal_queen8-1.smt2	✗	544	9	4680	**670**	740
jobshop6-2-3-3-2-4-9.smt2	✗	252	25	7288	**321**	>60000
jobshop6-2-3-3-2-4-12.smt2	✓	252	25	36378	295	**160**
super_queen9-1.smt2	✗	570	10	896	**128**	210
diamonds.11.3.i.a.u.smt2	✗	188	78	25526	**817**	>60000
toroidal_queen9-1.smt2	✗	738	10	14005	**2267**	3030
diamonds.16.2.i.a.u.smt2	✗	209	81	>60000	**49910**	>60000
diamonds.12.3.i.a.u.smt2	✗	205	85	57284	**1584**	>60000
diamonds.17.2.i.a.u.smt2	✗	222	86	>60000	>60000	>60000
toroidal_queen10-1.smt2	✗	880	11	>60000	**7346**	16490
super_queen11-1.smt2	✓	834	12	356	296	**90**
queen12-1.smt2	✓	816	13	752	521	**140**
jobshop8-2-4-4-4-4-12.smt2	✗	448	33	>60000	**184**	>60000
SortingNetwork4_live_bgmc002.smt2	✓	503	21	28	**17**	110
diamonds.10.5.i.a.u.smt2	✗	251	111	26782	**525**	>60000
inf-bakery-invalid-4.smt2	✓	536	23	**20**	46	48050
super_queen12-1.smt2	✓	984	13	988	260	**130**
queen14-1.smt2	✓	1120	15	11445	**1810**	1850
LinearSearch_live_bgmc003.smt2	✓	673	36	28	**0**	430
queen15-1.smt2	✓	1290	16	14397	3272	**2150**
DTP_k2_n35_c175_s4.smt2	✗	699	35	10317	**1076**	>60000
toroidal_queen13-1.smt2	✓	1586	14	3548	678	**630**
super_queen15-1.smt2	✓	1506	16	12109	**659**	5280
queen16-1.smt2	✓	1472	17	1728	767	**360**
inf-bakery-mutex-7.smt2	✗	914	38	**132**	288	>60000
super_queen16-1.smt2	✓	1704	17	50195	**3770**	12270
queen17-1.smt2	✓	1666	18	7876	1213	**850**

10. Conclusions and Future Work

This paper has presented a DPLL(T) SMT solver coded in Prolog for three theories – rational-tree unification, linear arithmetic and integer difference constraints. The motivation for this work is the need for an SMT solver over rational-tree unification in order to recover types from x86 binaries; with Prolog providing a decision procedure for rational-tree unification the integration with the SAT solver in [25] is a natural development. The effectiveness of the approach has been demonstrated by the successful application of the solver to a suite of type recovery problems.

The solver can be extended by providing decision procedures for further theories. Finite domain solvers, such as SICStus CLP(FD), often allow reified constraints [9], hence finite domain constraints might appear a good candidate to incorporate into the DPLL(T) framework. Unfortunately, finite domain constraint solvers typically maintain stores that are potentially inconsistent, hence without labelling (an unattractive step) a decision procedure for conjunctions of theory constraints is not readily available. That said, finite domain techniques have been applied to infer typings for the predicates of logic programs that are disjunctive [16]. Like our work this approach avoids the need for fixpoint computation, and its use of propagator constraints echoes theory propagation. However, this work assumes type definitions are prescribed up-front and recasting type recovery as SMT, with use of the core algorithm, can in principle resolve inconsistencies by applying MaxSMT. Future work will focus on this direction of inquiry.

The integer difference solver performs suprisingly well, and it would certainly be worthwhile investigating what further improvements could be made in order to tackle harder problems. The search and labelling heuristics described in section 4.2 provide a good starting point, and additional domain specific optimisations could also be made.

The approach to theory propagation described in this paper is not necessarily tied to DPLL-based SAT solvers and future work is to describe how to integrate it into a generalisation [53] of Stålmarck's proof procedure [51]. Other future work is to add certification, as in [1]. That is, for unsatisfiable instances not only is the result returned, but also a demonstration of unsatisfiability that can be determined by a small trusted computing base.

11. Acknowledgments

We thank Alan Mycroft for stimulating discussions on type recovery at Dagstuhl Seminar 12051 [32]. We also thank Wei-Ming Khoo for explaining his approach to type recovery and Mats Carlsson for his help with SICStus Prolog. Finally, we would like to thank both the PPDP and SCP reviewers for their helpful comments, and for alerting us to QuickXplain [30].

References

[1] S. Anoep, E. Drijver, A. Ganga, and M. Kirsten. Resolution Proof for Look-ahead SAT Solvers. 2006. `www.st.ewi.tudelft.nl/sat/reports/resolution.pdf`.

[2] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: an Open Platform for Satisfiability Modulo Theories. *Electronic Notes in Theoretical Computer Science*, 125(3):25–36, 2005.

[3] G. Balakrishnan and T. Reps. Recovery of Variables and Heap Structure in x86 Executables. Technical report, University of Wisconsin, 2005.

[4] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[5] C. Baykan and M. Fox. Spatial Synthesis by Disjunctive Constraint Satisfaction. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11(4):245–262, 1997.

[6] A. Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[7] R. Bruttomesso, D. Cok, and A. Griggio. SMT-COMP 2012. `http://smtcomp.sourceforge.net/2012/`, 2012.

[8] T. Bull, E. Younger, K. Bennett, and Z. Luo. Bylands: Reverse Engineering Safety-critical Systems. In *International Conference on Software Maintenance*, pages 358–366. IEEE Computer Society, 1995.

[9] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.

[10] D. R. Chase, M. N. Wegman, and K. F. Zadeck. Analysis of Pointers and Structures. In *Programming Language Design and Implementation*, pages 296–310. ACM Press, 1990.

[11] V. Chipounov and G. Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *European Conference on Computer Systems*, pages 167–180. ACM Press, 2010.

[12] M. Codish, V. Lagoon, and P. J. Stuckey. Logic Programming with Satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.

[13] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *Computer and Communications Security*, pages 391–402. ACM Press, 2008.

[14] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[15] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[16] B. Demoen, M. García de la Banda, and P. J. Stuckey. Type Constraint Solving for Parametric and Ad-hoc Polymorphism. In *Australian Computer Science Conference*, pages 217–228. Springer, 1999.

[17] W. Drabent. Logic + Control: An Example. In *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 301–311. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[18] R. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[19] B. Guha, C. Davis, and B. Mukherjee. Network Security via Reverse Engineering of TCP code: Vulnerability Analysis and Proposed Solutions. In *Conference on Computer Communications*, pages 603–610. IEEE Computer Society, 1996.

[20] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *International Conference on Logic Programming*, pages 631–645. MIT Press, 1995.

[21] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 260–268. Springer, 1992.

[22] J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2001.

[23] J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.

[24] J. M. Howe and A. King. A Pearl on SAT Solving in Prolog. In *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2010.

[25] J. M. Howe and A. King. A Pearl on SAT and SMT Solving in Prolog. *Theoretical Computer Science*, 435:43–55, 2012.

[26] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, . . . , ω.* PhD thesis, Université Paris VII, 1976.

[27] J. Jaffar. Efficient Unification Over Infinite Trees. *New Generation Computing*, 2(3):207–219, 1984.

[28] J. Jaffar, A. E. Santosa, and R. Voicu. An Interpolation Method for CLP Traversal. In *Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2009.

[29] R. G. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[30] U. Junker. QuickXPlain: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI*, pages 167–172. AAAI Press, 2004.

[31] D. Kapur. Shostak's Congruence Closure as Completion. In *Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 1997.

[32] A. King, A. Mycroft, T. W. Reps, and A. Simon. Analysis of Executables: Benefits and Challenges (Dagstuhl Seminar 12051). *Dagstuhl Reports*, 2(1):100–116, 2012.

[33] R. A. Kowalski. Algorithm = Logic + Control. *Communication of the ACM*, 22(7):424–436, 1979.

[34] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[35] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Network and Distributed System Security Symposium*. The Internet Society, 2011.

[36] C. M. Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 1997.

[37] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Network and Distributed System Security Symposium*. The Internet Society, 2010.

[38] D. MacQueen, G. Plotkin, and R. Sethi. An Ideal Model for Recursive Polymorphic Types. In *Principles of Programming Languages*, pages 165–174. ACM Press, 1983.

[39] M. J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Logic in Computer Science*, pages 348–357. IEEE Computer Society, 1988.

[40] B. P. Miller and A. R. Bernat. Anywhere, Any Time Binary Instrumentation. In *Workshop on Program Analysis for Software Tools and Engineering*, September 2011. See also `http://www.dyninst.org/dyninst`.

[41] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[42] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM Press, 2001.

[43] A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 1999.

[44] L. Naish. *Negation and Control in Logic Programs*, volume 238 of *Lecture Notes in Computer Science*. Springer, 1986.

[45] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In *Computer-Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005.

[46] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[47] D. Pham, J. Thornton, and A. Sattar. Modelling and Solving Temporal Reasoning as Propositional Satisfiability. *Artificial Intelligence*, 172(15):1752–1782, 2008.

[48] G. Ramalingam, J. Field, and F. Tip. Aggregate Structure Identification and Its Application to Program Analysis. In *Principles of Programming Languages*, pages 119–132. ACM Press, 1999.

[49] E. Robbins, J. M. Howe, and A. King. Theory Propagation and Rational-Trees. In *Principles and Practice of Declarative Programming*, pages 193–204. ACM Press, 2013.

[50] M. Sharif, A. Lanzi, J. Giffin, and L. Wenke. Automatic Reverse Engineering of Malware Emulators. In *Symposium on Security and Privacy*, pages 94–109. IEEE Computer Society, 2009.

[51] M. Sheeran and G. Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[52] O. Strichman, S. Seshia, and R. Bryant. Deciding Separation Formulas with SAT. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 2002.

[53] A. V. Thakur and T. W. Reps. A Generalization of Stålmarck's Method. In *Static Analysis Symposium*, volume 7460 of *Lecture Notes in Computer Science*, pages 334–351. Springer, 2012.

[54] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *European Conference on Logics in Artificial Intelligence*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer, 2002.

[55] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[56] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.