Theory Propagation and Rational-Trees

Ed Robbins
School of Computing
University of Kent
er209@kent.ac.uk

Jacob M. Howe
School of Informatics
City University London
jacob@soi.city.ac.uk

Andy King
School of Computing
University of Kent
a.m.king@kent.ac.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 closely 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 many Prolog systems. The solver is benchmarked against a number of type recovery problems, and compared against a lazy-basic SMT solver built on PicoSAT.

Categories and Subject Descriptors

[**Software notations and tools**]: Constraint and logic languages; [**Software and application security**]: Software reverse engineering; [**Semantics and reasoning**]: Program analysis

General Terms

Theory of Computation, Software and its Engineering

Keywords

SAT solving, reverse engineering

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 motived 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 [9, 13];
- Reuse of legacy software without source code for guaranteed compliance with hardware IO and/or timing behaviour, for example, for hardware drivers [7] or control systems [4];
- Understanding the operation of, and threat posed by, viruses and other malicious code by anti-virus companies [40].

An important problem in reverse engineering is that of type recovery [35]. 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 [31]. 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 [31], but its application to the recovery 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 [24, 37] which is strictly weaker than circular unification and cannot capture recursive types. Unfortunately the theory of rational-trees is not

PPDP '13, September 16 - 18 2013, Madrid, Spain

http://dx.doi.org/10.1145/2505879.2505901.

currently supported in any off-the-shelf SMT solver, hence this investigation into how to prototype 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 [8] 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 [14, 15]. (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 [26] 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 [36] can be used to succinctly specify the watched literal technique [34]. 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 [16, 17, 19]. In addition to issues of performance the correctness of this approach has been examined [12]. To summarise, Prolog not only provides constraint libraries, but also the facility to implement a succinct SAT solver [19]. 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 outperforms an SMT solver constructed from PicoSAT [3] and a Prolog coded theory solver using the lazy-basic approach.

1.4 Contributions

This paper shows how to integrate theory propagation and unit propagation in Prolog using reification and thereby realise an SMT solver in Prolog which can solve type recovery problems. 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 two theories. The first theory is that of rational-trees [32], where the control provided by block and when-declarations can realise reification. Efficient rational-tree unification [21] 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.

- 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 [29] 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 [2, 6, 28, 30, 35, 39], 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 [35].

```
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 simplifies 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 [10]. 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 [20], constraints describing unification of terms and type variables, are now derived for each instruction. These are similar to the disjunctive constraints described by [35] 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:

$$(T_{C_2} = \text{basic}(\_, \text{int}, 4) \wedge T_{C_1} = T_{C_2})$$

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

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, and 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	$(T_{A_2} = \text{basic}(\_,\ \text{int},\ 4) \land T_{A_1} = T_{A_2}) \lor \begin{pmatrix} T_{A_1} = \text{ptr}(\langle[\dots],\alpha_0,[\_,\_,\_,\alpha_1,\dots]\rangle) \land \\ T_{A_2} = \text{ptr}(\langle[\dots,\alpha_0,\_,\_,\_],\alpha_1,[\dots]\rangle) \end{pmatrix}$
3	$T_{A_2} = \text{ptr}(\langle[\dots],T_{edx_1},[\_,\_,\_,\dots]\rangle)$
4	$T_{eax_1} = \text{basic}(\_,\ \text{int},\ 4) \lor T_{eax_1} = \text{ptr}(\langle[\dots],\alpha_2,[\dots]\rangle)$
5	$(T_{eax_2} = T_{eax_1} \land T_{edx_2} = T_{edx_1}) \lor (T_{eax_2} = T_{eax_3} \land T_{edx_2} = T_{edx_3})$
8	$T_{edx_2} = \text{ptr}(\langle[\dots],T_{B_1},[\dots]\rangle)$
9	$\begin{pmatrix} T_{eax_3} = \text{basic}(\_,\text{int},4) \land \\ T_{eax_2} = T_{eax_3} \land T_{B_1} = T_{eax_3} \end{pmatrix} \lor \begin{pmatrix} T_{eax_3} = \text{ptr}(\langle[\dots],\alpha_3,[\dots]\rangle) \land \\ T_{eax_2} = \text{ptr}(\langle[\dots],\alpha_4,[\dots]\rangle) \land \\ T_{B_1} = \text{basic}(\_,\text{int},4) \end{pmatrix} \lor \begin{pmatrix} T_{eax_3} = \text{ptr}(\langle[\dots],\alpha_5,[\dots]\rangle) \land \\ T_{eax_2} = \text{basic}(\_,\text{int},4) \land \\ T_{B_1} = \text{ptr}(\langle[\dots],\alpha_6,[\dots]\rangle) \end{pmatrix}$
10	$T_{edx_2} = T_{C_1}$
11	$(T_{C_2} = \text{basic}(\_,\ \text{int},\ 4) \land T_{C_1} = T_{C_2}) \lor \begin{pmatrix} T_{C_1} = \text{ptr}(\langle[\dots],\alpha_7,[\_,\_,\_,\alpha_8,\dots]\rangle) \land \\ T_{C_2} = \text{ptr}(\langle[\dots,\alpha_7,\_,\_,\_],\alpha_8,[\dots]\rangle) \end{pmatrix}$
12	$T_{C_2} = \text{ptr}(\langle[\dots],T_{edx_3},[\_,\_,\_,\dots]\rangle)$

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

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

which requires rational-tree unification.

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, $\text{ptr}(\langle[\dots],\alpha_5,[\dots]\rangle)$. Both solutions have the following typings in common:

$$T_{B_1} = \text{basic}(\_,\text{int},4)$$

$$T_{edx_1} = T_{edx_2} = T_{edx_3} = T_{C_1} = \text{ptr}(\langle[\dots],\text{basic}(\_,\ \text{int},\ 4),[\_,\_,\_,T_{edx_1},\_,\_,\_,\dots]\rangle)$$

$$T_{C_2} = \text{ptr}(\langle[\dots,\text{basic}(\_,\text{int},4),\_,\_,\_],T_{edx_1},[\_,\_,\_,\dots]\rangle)$$

$$T_{A_2} = \text{ptr}(\langle[\dots,\alpha_0,\_,\_,\_],T_{edx_1},[\_,\_,\_,\dots]\rangle)$$

$$T_{A_1} = T_{esp_0} = \text{ptr}(\langle[\dots],\alpha_0,[\_,\_,\_,T_{edx_1},\_,\_,\_,\dots]\rangle)$$

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 [11] with watched literals [34]; this includes the solver in [18] 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 $f = (\neg x \lor z) \land (u \lor \neg v \lor w) \land (\neg w \lor y \lor \neg z)$ 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 $(\neg x \lor z)$ to be satisfiable, hence f as a whole, it is necessary that $z \mapsto \mathit{true}$. Moreover, for $(\neg w \lor y \lor \neg z)$ to be satisfiable, it follows that $w \mapsto \mathit{false}$. The satisfiability of $(u \lor \neg v \lor w)$ 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, $x \mapsto \mathit{false}$ is added to θ; if this fails search backtracks to a previous assignment. Further details can be found in [18, 44].

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 [38, 43]. 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 $(a < b) \land (a = 0 \lor a = 1) \land (b = 0 \lor b = 1) \models (a + b = 1)$ amounts to determining that the theory formula $\phi = (a < b) \land (a = 0 \lor a = 1) \land (b = 0 \lor b = 1) \land \neg(a+b = 1)$ is not satisfiable. For this problem, the set of literals is $\Sigma = \{a < b, \dots, a + b = 1\}$.

Suppose, in addition, that the encoder mapping is defined:

$$e(a < b)=x, \quad e(a = 0)=y, \quad e(a = 1)=z,$$
$$e(b = 0)=u, \quad e(b = 1)=v, \quad e(a + b = 1)=w$$

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. Where $\ell \in \Sigma$, a conjunct is 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 [19].

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 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) deriving of Algorithm 11.2.3 from [27]. 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 input the 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 [19]. 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 $\hat{Th}(\theta \cup \{e(\ell) \mapsto false\}, e)$

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 [19]. Search is then driven by assignments to the variables using `elim_vars`.

```
function DPLL(T)(f: CNF formula, θ : truth assignment, e : Σ → X)
begin
    (θ_3, res) := propagate(f, θ, e, ∅);
    if (is-satisfied(f, θ_3)) then
        return θ_3;
    else if (res = ⊥) then
        return ⊥;
    else
        x := choose-free-variable(f, θ_3);
        (θ_4, res) := DPLL(T)(f, θ_3 ∪ {x ↦ true}, e);
        if (res = ⊤) then
            return θ_4;
        else
            return DPLL(T)(f, θ_3 ∪ {x ↦ false}, e);
        endif
    endif
end
```

```
function propagate(f: CNF formula, θ : truth assignment, e : Σ → X, D : set of theory literals)
begin
    θ_1 := θ ∪ {e(ℓ) ↦ true | ℓ ∈ D ∩ Σ} ∪ {e(ℓ) ↦ false | ¬ℓ ∈ D ∧ ℓ ∈ Σ};
    θ_2 := θ_1 ∪ unit-propagation(f,θ_1);
    ⟨D, res⟩ := deduction(T̂h(θ_2, e));
    if (D = ∅ ∨ res = ⊥)
        return (θ_2, res);
    else
        return propagate(f, θ_2, e, D);
    endif
end
```

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

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

Figure 2: Interface to the DPLL(T) solver

Implementing the interface provided by predicates `setup` and `post_theory`, together with the SAT solver from [19] 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 [19] 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 explanation of labelling.

One classic strategy for labelling that is also straightforward to incorporate into a solver written in a declarative language is to rank variables by their number of occurrences in clauses of minimal size [23]. This associates a weight to each unbound variable according to the number of its 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 [29] 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 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 [29], 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.

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 [19] an algorithm adapted from [22] was used). Further, in the application 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) 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.

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).
```

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 [18].

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

function findcore ($e = [t_1 \mapsto x_1, \ldots, t_n \mapsto x_n] : \Sigma \to X$, f : CNF formula, c : int, $core : \Sigma \to X$)
begin
if ($e = [\,]$)
return $core$;
else if ($c = 0$)
$core' := [t_1 \mapsto x_1, t_n \mapsto x_n] \cup core$;
findcore($[t_2 \mapsto x_2, \ldots, t_{n-1} \mapsto x_{n-1}]$, f, $\lfloor \frac{n-1}{2} \rfloor$, $core'$);
else
$i := 1$; $j := n$;
if ($\neg$DPLL(T)(f, $\emptyset$, $[t_{c+1} \mapsto x_{c+1}, \ldots, t_n \mapsto x_n] \cup core$))
$i := c + 1$;
if ($\neg$DPLL(T)(f, $\emptyset$, $[t_i \mapsto x_i, \ldots, t_{n-c} \mapsto x_{n-c}] \cup core$))
$j := n - c$;
if ($c = 1$)
$c' := 0$;
else
$c' := \lfloor \frac{c+1}{2} \rfloor$;
endif
findcore($[t_i \mapsto x_i, \ldots, t_j \mapsto x_j]$, f, c', $core$);
endif
end

Figure 3: Finding an unsatisfiable core

```
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

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. EXPERIMENTAL RESULTS

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 the 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

```
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

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 [33] 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 paper 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 [3], 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. To clarify the meaning of the columns of Table 1, 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[1], indicated by seg, invariably after several hours of computation. The fault is repeatable.

In addition to these timing results, the recursive types inferred for mergesort, 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. DISCUSSION

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.

[1]This bug has been fixed in the forthcoming SICStus 4.3.

Table 1: Benchmarking for a selection of type recovery problems

benchmark[a]	instruc[b]	clauses[c]	prop-vars[d]	theory-vars[e]	time[f]	SMT calls[g]	time[h]	SAT calls[i]
1 iterative-sum.O1	296	2047	564	779	14.57	(SAT) 1	413.36	796
2 iterative-sum.O2	312	2132	586	812	52.34	(SAT) 1	seg	
3 recursive-sum.O1	302	2129	588	809	15.37	(SAT) 1	6382.50	998
4 mergesort.O0	480	3216	888	1220	585.89	70	seg	
5 mergesort.O1	387	2636	718	1011	20.05	(SAT) 1	1176.58	1720
6 mergesort.O2	395	2628	713	1017	20.30	(SAT) 1	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	
84 mbslen	25895	257132	35148	148102	1935.12	56	>14400	

[a] the name of the binary from which the constraints were generated
[b] the number of instructions in the binary, over which the constraints were drawn
[c] the number of propositional clauses in the problem
[d] the number of propositional variables
[e] the number of theory variables
[f] the runtime in seconds to find a model or a core for the Prolog solver
[g] the number of times the Prolog SMT solver was called
[h] the runtime in seconds to find a model or a core for the hybrid solver
[i] the number of times the PicoSAT propositional solver was called

This explains why it is relatively slow. Benchmark 7 has timed out, a reminder that large SMT problems can be hard to solve.

Note that the time require 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** $\mathbf{ebx_1}$, 0x605e38 appears in the cksum binary, and moves the address of a string into $\mathbf{ebx_1}$ 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. CONCLUSIONS AND FUTURE WORK

This paper has presented a DPLL(T) SMT solver coded in Prolog for two theories – rational-tree unification and quantifier-free linear real arithmetic. 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 [19] 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 [5], 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.

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 [42] of Stålmarck's proof procedure [41]. 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. Another line of inquiry will be to investigate how to systematically relax the SMT instance so that a type assignment can be always found, without manual intervention, even in the presence of conflicting constraints. MaxSMT techniques seem to be well-suited to this task.

10. ACKNOWLEDGMENTS

We thank Alan Mycroft for stimulating discussions on type recovery at Dagstuhl Seminar 12051 [25]. We also thank Wei-Ming Khoo for explaining his approach to type recovery and Mats Carlsson for his help with SICStus Prolog.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

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

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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`.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.