Automatic Type Inference via Partial Evaluation

Aaron Tomb
University of California, Santa Cruz
1156 High St.
Santa Cruz, CA 95060
atomb@cs.ucsc.edu

Cormac Flanagan
University of California, Santa Cruz
1156 High St.
Santa Cruz, CA 95060
cormac@cs.ucsc.edu

ABSTRACT

Type checking and type inference are fundamentally similar problems. However, the algorithms for performing the two operations, on the same type system, often differ significantly. The type checker is typically a straightforward encoding of the original type rules. For many systems, type inference is performed using a two-phase, constraint-based algorithm.

We present an approach that, given the original type rules written as clauses in a logic programming language, automatically generates an efficient, two-phase, constraint-based type inference algorithm. Our approach works by partially evaluating the type checking rules with respect to the target program to yield a set of constraints suitable for input to an external constraint solver. This approach avoids the need to manually develop and verify a separate type inference algorithm, and is ideal for experimentation with and rapid prototyping of novel type systems.

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms

Languages, Experimentation

Keywords

Type Systems, Logic Programming, Applications of Declarative Programming, Program Analysis

1. INTRODUCTION

Type systems provide numerous benefits in terms of software reliability, performance, and maintainability. Type systems are typically specified via a collection of type rules. If these type rules are syntax-directed (*i.e.*, there is exactly one type rule for each language construct), then deriving a corresponding type checking algorithm is straightforward.

However, developing a corresponding type inference algorithm is often significantly more challenging. A number of type inference (as well as program analysis) problems can be efficiently solved via a two-phase approach. First, a *constraint-generating* phase traverses the abstract syntax tree of the target program to generate a collection of constraints over the type variables. Second, a *constraint-solving* phase then solves these constraints. In addition to the difficulties of developing such two-phase algorithms, validating their correctness is non-trivial, since it requires formalizing the semantics of the intermediate constraint language as well as specifying the behavior of both analysis phases. In particular, specifying the constraint-generating phase often requires a separate set of *constraint-generating* type rules [5].

In this paper, we propose an approach that avoids these difficulties and complexities by using partial evaluation to automatically derive an efficient, two-phase type inference algorithm from the original type rules. We assume that the type rules are represented as clauses in a logic programming language such as Prolog. Because the Horn clauses in logic programming closely resemble type rules, we can directly convert existing type rules to Prolog clauses in a straightforward and elegant manner.

Type checking using the Prolog clauses is trivial. We simply need to pass the program and its associated types into the predicate associated with whole-program judgement. If the type rules are syntax-directed, a conclusion can be reached in an efficient and deterministic manner.

Due to the powerful search mechanism inherent in logic programming, type inference is almost as simple (although quite inefficient). If we use the same Prolog code, but leave type variables unbound, the depth-first search performed by the Prolog implementation will attempt to find bindings that satisfy the type rules. Thus, the Prolog encoding of the type rules can serve the needs of both checking and inference in an elegant and flexible manner.

However, while the Prolog implementation of the type rules suffices for efficient type checking, Prolog's depth-first search strategy is extremely inefficient for many type inference problems. In particular, as we show in Section 2, it may diverge even for comparatively simple type systems for which efficient type inference algorithms exist. Thus, the Prolog clauses provide a clear specification but extremely inefficient implementation of a type inference algorithm.

PPDP'05, July 11–13, 2005, Lisbon, Portugal.

The key contribution of this paper is to demonstrate how to automatically translate the type rules (represented as Prolog clauses) into an efficient, two-phase, constraint-based type inference algorithm. The essence of our approach is to *partially evaluate* these type rules. Apart from these type rules (represented as Prolog clauses), the partial evaluator only requires as input a simple *partitioning parameter* that specifies how to partition the type inference computation between the two phases. More specifically, the partial evaluator essentially interprets the Prolog type checking clauses, and the partitioning parameter specifies which parts of this computation should be delayed until the second, constraint-solving phase. Typically, any relation that depends on the actual types associated with type variables is delayed until the second phase, which find a solution for these type variables.

In general, the result of the first, partial-evaluation phase, called the *residual constraint*, may be an arbitrary combination of delayed Prolog terms. However, for many type systems, the natural partitioning of the computation, as might be used in a hand-coded two-phase analysis, yields a simpler constraint language that can be solved efficiently. Such constraint languages include propositional satisfiability and Datalog, for which external solvers can find solutions much more efficiently than a standard Prolog implementation.

We have validated this approach on a number of type systems and program analyses. Our experience indicates that, in comparison to the traditional approach of using manual effort to develop, debug, and verify type inference algorithms, our proposed approach has a number of key advantages:

- It provides a method for automatically deriving an efficient, two-phase type inference algorithm from the original type checking or analysis rules.
- It removes the need to prove the correctness of the inference algorithm, since its correctness follows from the correctness of the partial evaluator.
- The Prolog representation of the type rules also functions as an efficient type checker.

We note that, for the common case where the type checking rules are syntax-directed, type checking and constraint generation are deterministic and efficient, and partial evaluation yields a conjunction of delayed terms or constraints. On the other hand, if the type checking rules are not syntax-directed, then type checking and constraint generation are not deterministic, and the residual constraint may include both conjunctions and disjunctions of constraints. Thus, our approach is still applicable, but may be less efficient.

The remainder of this paper is organized as follows. Section 2 and Section 3 illustrate our approach, including the implementation details. Section 4 describes experiments with larger languages and type systems, and some work in progress. Finally, Section 6 describes related work and Section 7 gives some concluding remarks and describes our ideas for future work.

2. TYPE INFERENCE FOR SUBTYPING

To illustrate our approach to automatic type inference, this section presents an example of a simple type system with subtyping over base types. The λ_{sub} language extends the simply-typed λ-calculus with numeric constants, addition, multiplication, and conditional expressions, as shown in Figure 1.

For $x \in$ Identifiers, $n_i \in \mathbb{Z}$, $n_r \in \mathbb{R} - \mathbb{Z}$.

$$\begin{array}{rcl} e & ::= & n_i \\ & | & n_r \\ & | & x \\ & | & e_1\ e_2 \\ & | & \lambda x : t.e \\ & | & \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \\ & | & e_1\ \mathit{op}\ e_2 \\ \mathit{op} & ::= & + \mid * \\ t & ::= & \mathit{base}(b) \mid t \to t \\ b & ::= & \mathit{int} \mid \mathit{real} \end{array}$$

Figure 1: Syntax for λ_{sub}.

2.1 Type Checking

The type system for λ_{sub} includes a notion of subtyping between the base types *int* and *real*, where *int* is a subtype of *real*. The full type rules are shown in Figure 2. These type rules are syntax-directed, so type checking takes time linear in the size of the input program. (Many of the type rules are directed by the syntax of expressions, and the rules defining the subtyping relation are directed by the syntax of types.)

Figure 3 shows a straightforward Prolog implementation of a type checker for this type system. Each rule in the type system corresponds to a clause in the Prolog code, and each judgement corresponds to a predicate.

To illustrate the type checking process, consider the following program:

$$P \ \equiv\ (\lambda x : \mathit{base}(\mathit{int}).\ x + 1)$$

We can pass P (converted to Prolog syntax) into the type checker with the following goal, where `[]` represents the empty environment:

`tc([], ` P`, ` T`).`

Because the definition of P provides a type for the argument of the λ abstraction, type checking is syntax-directed and the Prolog code executes deterministically, with no backtracking. As expected, execution of this goal causes T to be bound to the type:

$$\mathit{base}(\mathit{int}) \to \mathit{base}(\mathit{int})$$

2.2 Type Inference

The Prolog representation of the type rules gives more than elegant type checking code, however — it also provides a high-level executable specification of a type inference algorithm.

Consider the unannotated version of the program from Section 2.1:

$$P' \equiv (\lambda x : T_x.\ x + 1)$$

$\boxed{b \sqsubseteq b}$

$$\frac{}{b \sqsubseteq_b b}$$

$$\frac{}{int \sqsubseteq_b real}$$

$\boxed{t \sqsubseteq t}$

$$\frac{b_1 \sqsubseteq_b b_2}{base(b_1) \sqsubseteq base(b_2)}$$

$$\frac{\begin{array}{c} t_1' \sqsubseteq t_2 \\ t_2 \sqsubseteq t_2' \end{array}}{t_1 \rightarrow t_2 \sqsubseteq t_1' \rightarrow t_2'}$$

$\boxed{B \vdash e : t}$

$$\frac{}{B \vdash n_i : base(int)}$$

$$\frac{}{B \vdash n_r : base(real)}$$

$$\frac{}{B, x : t, B' \vdash x : t}$$

$$\frac{\begin{array}{c} B \vdash e_1 : t_1 \rightarrow t \\ B \vdash e_2 : t_2 \\ t_2 \sqsubseteq t_1 \end{array}}{B \vdash e_1\ e_2 : t}$$

$$\frac{B, x : t_1 \vdash e : t_2}{B \vdash \lambda x : t_1.e : t_1 \rightarrow t_2}$$

$$\frac{\begin{array}{c} B \vdash e_1 : base(int) \\ B \vdash e_2 : t_2 \\ B \vdash e_3 : t_3 \\ t_2 \sqsubseteq_b t, t_3 \sqsubseteq_b t \end{array}}{B \vdash \mathit{if}\ e_1\ then\ e_2\ else\ e_3 : t}$$

$$\frac{\begin{array}{c} B \vdash e_1 : base(t_1) \\ B \vdash e_2 : base(t_2) \\ t_1 \sqsubseteq t, t_2 \sqsubseteq t \end{array}}{B \vdash e_1\ op\ e_2 : base(t)}$$

Figure 2: Type rules for λ_{sub}.

Grammar for the Prolog representations:

```
E ::= int(N)
    | real(N)
    | var(X)
    | apply(E1, E2)
    | lambda(X, T, E)
    | if(E1, E2, E3)
    | op(Op, E1, E2)
Op ::= '+' | '*'
T ::= base(B)
    | func(t, t)
B ::= int | real
```

Prolog code:

```
base_subtype(B, B).
base_subtype(int, real).

subtype(base(B1), base(B2)) :-
    base_subtype(B1, B2).

subtype(func(A1, B1), func(A2, B2)) :-
    subtype(A2, A1),
    subtype(B1, B2).

tc(_, int(_), base(int)).

tc(_, real(_), base(real)).

tc(B, var(X), T) :-
    member(bind(X, T), B).

tc(B, apply(E1, E2), T) :-
    tc(B, E1, func(T1, T)),
    tc(B, E2, T2),
    subtype(T2, T1).

tc(B, lambda(X, T1, E), func(T1, T2)) :-
    tc([bind(X, T1)|B], E, T2).

tc(B, if(E1, E2, E3), T) :-
    tc(B, E1, base(int)),
    tc(B, E2, T2),
    tc(B, E3, T3),
    subtype(T2, T),
    subtype(T3, T).

tc(B, op(_, E1, E2), base(B3)) :-
    tc(B, E1, base(B1)),
    tc(B, E2, base(B2)),
    subtype(base(B1), base(B3)),
    subtype(base(B2), base(B3)).
```

Figure 3: Prolog implementation of the type rules for λ_{sub}.

where T_x is an unknown type, and is represented by an unbound Prolog variable. Though we leave T_x unbound, the Prolog representation of the type rules will still execute, but they no longer are deterministic or syntax-directed. Finding a solution for the goal:

$$\texttt{tc([], } P'\texttt{, } T\texttt{).}$$

now requires performing a *search* for types T_x and T that satisfy the type rules. In general, this search may be excessively expensive. In particular, due to Prolog's depth-first search algorithm, it may not terminate, even for this basic type system.

From the standpoint of simplicity and elegance, we have succeeded thus far: in only 22 lines of code derived directly from the type rules, we have provided an algorithm that performs both type checking and type inference. From a performance and termination point of view, on the other hand, type inference by direct execution (that is, depth-first search) of Prolog code is clearly inadequate.

In contrast, a hand-coded type inference algorithm can solve this problem in linear time using a two-phase constraint-based approach. The first phase walks over the program's abstract syntax tree to generate a set of constraints over the type variables. The second phase uses a separate solver that is tuned to the constraint class at hand to efficiently solve the constraints generated in the first phase.

The key insight of our work is that we can automatically generate an efficient two-phase constraint-based type inference algorithm. This approach requires as input only:

1. the Prolog encoding of the type checking rules, and
2. an additional *partitioning* parameter that describes how to partition the computation between the two phases.

We perform this automatic partitioning using partial evaluation. While many partial evaluators for Prolog already exist [21, 14, 15], they seem more complex then needed for our purposes. Instead, we wrote our own partial evaluator to provide precise control over which operations to evaluate, and which to delay.

Our partial evaluator (shown in Appendix A) consists of a simple meta-interpreter augmented with the ability to delay the evaluation of certain goals to the second, constraint-solving phase. The partial evaluator then returns a Boolean combination of these delayed goals, which we will call a *residual constraint*. Any solution to this residual constraint is then a solution to the original type inference problem, and vice versa. For clarity, our partial evaluator is restricted to a subset of Prolog that is adequate to express the type systems discussed in this paper. It could easily be extended to support other language constructs.

We consider our partial evaluator to be correct if solutions to the residual constraint it generates are also solutions to the original, directly-evaluated program. The following states this property more formally.

PROPERTY 1. *If θ is an assignment, tc is an algorithm, P is an input for tc, peval is our partial evaluator, and C is a residual constraint, then the following are equivalent:*

1. $\theta \models tc(P)$

2. $C = peval(tc(P))$ *and* $\theta \models C$

The format of the residual constraint returned by the partial evaluator varies depending on the original type rules and the partitioning parameter. In general, the residual constraint may be an arbitrary Prolog term. If the rules, in the presence of types, are purely syntax-directed, then the residual constraint will be a conjunction of the delayed terms. Non-syntax-directed rules can produce arbitrary conjunctions and disjunctions of delayed terms.

However, while residual constraints consisting of full Prolog are possible, in many situations the natural partitioning of the problem, as might be used in a hand-coded two-phase analysis, yields simple constraint languages (such as Datalog programs or Boolean expressions) for which efficient solving algorithms exist.

2.3 Base Type Inference

We initially focus on only inferring base types (that is, *int* vs *real*) for λ_{sub}. In this case, the residual constraint after partial evaluation is a conjunction of implications, which can be solved in linear time by a monotonic propositional satisfiability solver.

To illustrate the inference process more specifically, consider the following program:

$$P \;\equiv\; (\lambda z : base(b_1).\; (\lambda x : base(b_2).\; \textit{if } x \textit{ then } 3.14 \textit{ else } 0)\; ((\lambda y : base(b_3).\; y)\; z))$$

The *partitioning* parameter:

```
delay(base_subtype(_, _)).
```

tells the partial evaluator to postpone evaluation of any subtyping judgment between base types. This is the only constraint in the type rules that cannot be evaluated without knowledge of the actual base types of the type variables.[1]

Partially evaluating the type checker on this program via the goal:

```
peval(tc([], P, T), C).
```

yields the solution where T is $base(b_1) \rightarrow base(b_4)$, and the residual constraint C is a conjunction of the following base subtyping constraints:

$$\begin{aligned} real &\sqsubseteq_b b_{temp} \\ int &\sqsubseteq_b b_{temp} \\ b_{temp} &\sqsubseteq_b b_4 \\ b_1 &\sqsubseteq_b b_3 \\ b_3 &\sqsubseteq_b b_2 \\ b_2 &\sqsubseteq_b int \end{aligned}$$

We translate these constraints into a Boolean formula via the encoding

$$\begin{aligned} int &\equiv \texttt{false} \\ real &\equiv \texttt{true} \\ b_1 \sqsubseteq_b b_2 &\equiv b_1 \Rightarrow b_2 \end{aligned}$$

Because we are only inferring base types, the resulting formula is a monotonic satisfiability problem and can be solved in linear time, yielding the following minimal (*i.e.*, most precise) solution:

$$\begin{aligned} b_1 &= int \\ b_2 &= int \\ b_3 &= int \\ b_4 &= real \\ b_{temp} &= real \end{aligned}$$

which means that T is $base(int) \rightarrow base(real)$.

Thus, partial evaluation allows us to use a natural Prolog encoding of the type rules to perform both type checking and base type inference efficiently. We can directly execute the code for type checking, or use a partial evaluator to automatically partition the code into a two-phase algorithm for efficient base type inference.

[1] In general, this partitioning parameter describes which operations in the program depend on the actual bindings of the types being inferred, and thus should be delayed. This set of operations can exclude equality constraints, if desired, because the unification algorithm used in Prolog can keep track of equality without knowing the bindings of the variables in question. Excluding equality comparisons from the set of delayed operations leads to smaller generated constraint sets.

2.4 Full Type Inference

We next consider the more challenging problem of performing full type inference. For this problem, we need to instead delay the full `subtype` relation, via the partitioning parameter:

```
delay(subtype(_, _))
```

For example, consider the elaborate version of the identity function:

$$P \;\equiv\; (\lambda z : t_z.(\lambda x : t_x.x)\ z)(\lambda y : t_t.y)$$

Partially evaluating the type checker on this program via the goal:

```
peval(tc([], P, T), C).
```

produces the solution where $T = t_x$ and the residual constraint C is the conjunction of the following subtyping constraints:

$$t_z \sqsubseteq t_x$$
$$t_y \rightarrow t_y \sqsubseteq t_z$$

As expected, these constraints indicate that T is a supertype of $t_y \rightarrow t_y$, where t_y is unconstrained.

Performing full type inference introduces the full subtyping relation into the constraint language. In this case, the mapping to a monotonic satisfiability problem is no longer possible. However, the resulting constraints can still be solved efficiently, for example, via a constraint solving framework such as BANE [1] or BANSHEE [13].

Thus, the Prolog encoding of the original syntax-directed type rules is quite flexible – in addition to functioning as an efficient type checker, it can also (via appropriate partial-evaluation) yield a constraint-generator for both base type inference and full type inference.

3. SIGN ANALYSIS

To indicate the broad applicability of our approach, we next apply it to a significantly different analysis system. We consider the analysis of signs for a simple first-order language FOL.

A program in FOL consists of a number of function definitions, and a single expression, as described in Figure 4. Functions all take a single parameter, for simplicity, so the environment is either empty or consists of one binding, as described by the following grammar fragment. (The language could easily be extended to include multi-arity functions.)

$$B ::= \epsilon \mid x : s$$

The sign analysis rules for FOL are shown in Figure 5. In these rules, the domain for sign variables is the power set $2^{\{-,0,+\}}$, so each sign variable is a set that indicates what possible signs an expression can take. The rules for sign propagation through arithmetic operations and conditional expressions appear in Table 1 and Table 2, respectively.

These rules define a context-sensitive analysis of the possible signs of each function. Context-sensitivity [11] makes the analysis more precise, by giving each function a different input and output sign type in each context. It also makes the analysis more expensive, because we must reanalyze a function every time it appears in an expression.

For $x \in$ Identifiers, $n \in \mathbb{Z}$, $s \in 2^{\{-,0,+\}}$.

$$\begin{aligned}
P &::= d^* e \\
d &::= f(x) = e; \\
e &::= n \\
&\quad\mid x \\
&\quad\mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \\
&\quad\mid e_1\ \mathit{op}\ e_2 \\
&\quad\mid f(e) \\
\mathit{op} &::= + \mid *
\end{aligned}$$

Figure 4: Syntax for FOL.

		Result	
Operand 1: s_1	**Operand 2**: s_2	$+$	$*$
$\{-\}$	$\{-\}$	$\{-\}$	$\{+\}$
$\{-\}$	$\{0\}$	$\{-\}$	$\{0\}$
$\{-\}$	$\{+\}$	$\{-,0,+\}$	$\{-\}$
$\{0\}$	$\{-\}$	$\{-\}$	$\{0\}$
$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$
$\{0\}$	$\{+\}$	$\{+\}$	$\{0\}$
$\{+\}$	$\{-\}$	$\{-,0,+\}$	$\{-\}$
$\{+\}$	$\{0\}$	$\{+\}$	$\{0\}$
$\{+\}$	$\{+\}$	$\{+\}$	$\{+\}$

Table 1: Operator definition for $opsign(op, s_1, s_2)$. Each row defines an implication of set constraints. If a and b are the first two columns of row n, in order, and c is the appropriate result column for the operation in question, then row n defines the implication $a \subseteq s_1 \wedge b \subseteq s_2 \Rightarrow c \subseteq result$.

For a concrete example, consider the following program:

```
f(x) = g(x) + g(-2);
g(x) = x + 0;
f(0);
```

Under a context-sensitive analysis, the signs of `g(0)` and `g(-2)` are $\{0\}$ and $\{-\}$, respectively. Hence, the sign of the function call `f(0)` is a combination of these two signs, according to Table 1, yielding the resulting sign of $\{-\}$. If we performed the analysis in a context-insensitive manner, we would get the more approximate result $\{-,0\}$.

As in the λ_{sub} case, we can express the analysis rules in a natural manner as the Prolog code in Figure 6. The order of the `imply` clauses at the top of the program ensure that execution under Prolog's standard depth-first search semantics yields a minimal (most precise) solution. The `checkc` predicate in the Prolog code corresponds to the typing judgement $P \vdash f : s \rightarrow s'$. This predicate defines a relation associating a function f and an argument sign s with a result sign s', in the context of a given program P.

We can perform the sign analysis of a program by directly running the analysis code with the program to be analyzed as input. However, since the analysis rules and corresponding Prolog clauses are not syntax-directed, this approach results in an inefficient search and extremely poor performance. In particular, each function will be reanalyzed for each call site, even if it has previously been analyzed with the same sign parameter. Another issue is that, in the presence

Guard: s_1	Then: s_2	Else: s_3	Result
$\{+\}$	$\{+\}$	$\emptyset$	$\{+\}$
$\{+\}$	$\{0\}$	$\emptyset$	$\{0\}$
$\{+\}$	$\{-\}$	$\emptyset$	$\{-\}$
$\{0\}$	$\emptyset$	$\{+\}$	$\{+\}$
$\{0\}$	$\emptyset$	$\{0\}$	$\{0\}$
$\{0\}$	$\emptyset$	$\{-\}$	$\{-\}$
$\{-\}$	$\{+\}$	$\emptyset$	$\{+\}$
$\{-\}$	$\{0\}$	$\emptyset$	$\{0\}$
$\{-\}$	$\{-\}$	$\emptyset$	$\{-\}$

Table 2: Operator definition for $ifsign(s_1, s_2, s_3)$. Each row defines an implication of set constraints. If a, b, c and d are the four columns of row n, in order, then row n defines the implication $a \subseteq s_1 \wedge b \subseteq s_2 \wedge c \subseteq s_3 \Rightarrow d \subseteq result$.

$\boxed{\vdash P : s}$

$$\frac{\begin{array}{c} P = d^* e \\ P; \epsilon \vdash e : s \end{array}}{\vdash P : s}$$

$\boxed{P; B \vdash e : s}$

$$\frac{n < 0}{P; B \vdash n : \{-\}}$$

$$\frac{n = 0}{P; B \vdash n : \{0\}}$$

$$\frac{n > 0}{P; B \vdash n : \{+\}}$$

$$\frac{}{P; x\ s \vdash x : s}$$

$$\frac{\begin{array}{c} P; B \vdash e_1 : s_1 \\ P; B \vdash e_2 : s_2 \\ s = opsign(op, s_1, s_2) \end{array}}{P; B \vdash e_1\ op\ e_2 : s}$$

$$\frac{\begin{array}{c} P; B \vdash e_1 : s_1 \\ P; B \vdash e_2 : s_2 \\ P; B \vdash e_3 : s_3 \\ s = ifsign(s_1, s_2, s_3) \end{array}}{P; B \vdash if\ e_1\ then\ e_2\ else\ e_3 : s}$$

$$\frac{\begin{array}{c} P; B \vdash e : s \\ P \vdash f : s \rightarrow s' \end{array}}{P; B \vdash f(e) : s'}$$

$\boxed{P \vdash f : s \rightarrow s'}$

$$\frac{\begin{array}{c} f(x) = e \in P \\ P; x : s \vdash e : s' \end{array}}{P \vdash f : s \rightarrow s'}$$

Figure 5: Sign analysis rules for FOL.

```
imply(0, _, _).

imply(_, 0, _).

imply(1, 1, 1).

equal(S, S).

opsign('+', s(M1, Z1, P1), s(M2, Z2, P2),
        s(M3, Z3, P3)) :-
    imply(Z1, Z2, Z3), imply(Z1, M2, M3),
    imply(Z1, P2, P3), imply(M1, Z2, M3),
    imply(M1, M2, M3), imply(M1, P2, M3),
    imply(P1, Z2, P3), imply(P1, M2, M3),
    imply(P1, M2, P3), imply(M1, P2, P3),
    imply(P1, P2, P3), imply(P1, M2, Z3),
    imply(M1, P2, Z3).

opsign('*', s(M1, Z1, P1), s(M2, Z2, P2),
        s(M3, Z3, P3)) :-
    imply(Z1, Z2, Z3), imply(Z1, M2, Z3),
    imply(Z1, P2, Z3), imply(M1, Z2, Z3),
    imply(M1, M2, P3), imply(M1, P2, M3),
    imply(P1, Z2, Z3), imply(P1, M2, M3),
    imply(P1, P2, P3).

ifsign(s(GM, GZ, GP), s(TM, TZ, TP), s(FM, FZ, FP),
        s(M, Z, P)) :-
    imply(GM, TM, M), imply(GM, TZ, Z),
    imply(GM, TP, P), imply(GP, TM, M),
    imply(GP, TZ, Z), imply(GP, TP, P),
    imply(GZ, FM, M), imply(GZ, FZ, Z),
    imply(GZ, FP, P).

checkp(P, S) :-
    P = p(_, E),
    equal(AM, 0), equal(AZ, 0), equal(AP, 0),
    tc(P, s(AM, AZ, AP), E, S).

tc(_, _, num(N), s(1, 0, 0)) :-
    N < 0.

tc(_, _, num(N), s(0, 1, 0)) :-
    N = 0.

tc(_, _, num(N), s(0, 0, 1)) :-
    N > 0.

tc(_, B, var(X), S) :-
    member(bind(X,S),B).

tc(P, B, op(Op, E1, E2), S) :-
    tc(P, B, E1, S1),
    tc(P, B, E2, S2),
    opsign(Op, S1, S2, S).

tc(P, B, if(E1, E2, E3), S) :-
    tc(P, B, E1, S1),
    tc(P, B, E2, S2),
    tc(P, B, E3, S3),
    ifsign(S1, S2, S3, S).

tc(P, B, call(N, E), S) :-
    tc(P, B, E, S1),
    checkc(P, N, S1, S).

checkc(P, F, AS, RS) :-
    P = p(Defs, _),
    member(func(F, X, E), Defs),
    tc(P, [bind(X,AS)], E, RS).
```

Figure 6: Prolog implementation of the type rules for FOL.

of recursive functions, the Prolog derivation or proof tree can become infinite, though it will remain regular. Under standard Prolog semantics, the analysis of recursive functions would not converge.

We solve of both of these problems by using partial evaluation to partition the problem into two phases, where the residual goal of the first phase is a Datalog program that is then solved by the second phase, possibly via efficient techniques such as binary decision diagrams [24]. Our approach proceeds as follows:

1. We delay the implication that occurs in the definitions of the arithmetic and conditional operators, since this implication is dependent on the signs of variables, which are not known at partial-evaluation time. This is achieved with the following partitioning parameter:

   ```
   delay(imply(_, _, _)).
   ```

2. Since the `checkc` predicate is defined recursively, partial evaluation of the type checker may not terminate on recursive target programs unless this predicate is also delayed. However, we only *partially-delay* this predicate. That is, we retain the `checkc` predicate in the residual program, but in partially-evaluated form. For each function f in the target program, the residual program contains a clause for `checkc(P,`f`,`s`,`s'`)`, where the body of this clause has already been partially evaluated.

   In addition, the first parameter `P` is redundant, since the abstract syntax tree traversal has already been performed, and so this parameter is elided. Therefore, the partial evaluator transforms `checkc` to a relation on the final three parameters, whose body consists of the residual goal from the partial evaluation of its original body.

   We indicate that we want the body of a predicate to remain in the residual program in this fashion with the following `partdelay` partitioning parameter parameter:

   ```
   partdelay(checkc(_, _, _, _), [y,n,n,n]).
   ```

   where the list appearing after the predicate specification describes which parameters to remove — `y` means to remove the parameter in that position, and `n` means to retain it.

3. To give us a top-level predicate in the residual Datalog program, we delay the `checkp` predicate with:

   ```
   partdelay(checkp(_, _), [y,n]).
   ```

   where we again remove the first parameter, `P`, because it will never appear in the residual goal derived from the partial evaluation of the body.

After partial evaluation, a simple post-processing step can flatten all of the sign structures into three individual variables, yielding a Datalog program. The Datalog program can then be evaluated much more efficiently than an arbitrary Prolog program, because every Datalog relation is over a finite domain. This restriction allows for efficient execution strategies such as the use of binary decision diagrams [24].

Consider again the example program P from the beginning of this section. Partial evaluation with the goal:

```
peval(checkp(P, s(M, Z, P)), R)
```

followed by flattening gives that `R` is the following Datalog program:

```
imply(0, _, _).
imply(_, 0, _).
imply(1, 1, 1).

checkp(M, Z, P) :-
    checkc(f, 0, 1, 0, M, Z, P).

checkc(g, AM, AZ, AP, RM, RZ, RP) :-
    imply(AZ, 1, RZ), imply(AZ, T1, RM),
    imply(AZ, T2, RP), imply(AM, 1, RM),
    imply(AM, T1, RM), imply(AM, T2, RM),
    imply(AP, 1, RP), imply(AP, T1, RM),
    imply(AP, T1, RP), imply(AM, T2, RP),
    imply(AP, T2, RP), imply(AP, T1, RZ),
    imply(AM, T2, RZ).

checkc(f, AM, AZ, AP, RM, RZ, RP) :-
    checkc(g, AM, AZ, AP, GM1, GZ1, GP1),
    checkc(g, 1, 0, 0, GM2, GZ2, GP2),
    imply(GZ1, GZ2, RZ), imply(GZ1, GM2, RM),
    imply(GZ1, GP2, RP), imply(GM1, GZ2, RM),
    imply(GM1, GM2, RM), imply(GM1, GP2, RM),
    imply(GP1, GZ2, RP), imply(GP1, GM2, RM),
    imply(GP1, GM2, RP), imply(GM1, GP2, RP),
    imply(GP1, GP2, RP), imply(GP1, GM2, RZ),
    imply(GM1, GP2, RZ).
```

The goal `checkp(M, Z, P)` has the minimal solution: `M = 1`, `Z = 0`, `P = 0`. Or, in other words, the expression `f(0)` has the sign $\{-\}$, as expected.

This section has shown that partial evaluation can effectively produce constraints in forms other than that of Boolean expressions, but which also can be solved in a reasonably efficient manner.

4. APPLICATIONS AND EXPERIMENTS

4.1 Sign Analysis for Java

For clarity, the previous detailed examples focused on idealized languages and type systems, to most clearly illustrate the ideas of this paper. However, the methods we have described do scale to realistic languages. As an initial example, we implemented a sign analysis system for the Java programming language [12].

First, we developed a translator, based on the front-end Javafe [6], that converts Java source code into a structured Prolog term representing the program's abstract syntax tree. Then we implemented a set of rules for a context-insensitive sign analysis, in about 300 lines of Prolog code, that is modeled on the analysis shown in Figure 5.

We can execute this Prolog code directly, but execution time becomes prohibitively high for large programs. Alternatively, we can partially evaluate the Prolog code with respect to a particular target program to produce constraints that are solvable by a monotonic satisfiability solver. For these experiments, the Chaff solver [17] yielded excellent (essentially linear time) performance, even though it is not specifically focused on monotonic constraints.

To empirically evaluate the benefit of our approach, we applied both forms of the analysis to a number of Java programs of various sizes. Figure 7 compares the time taken

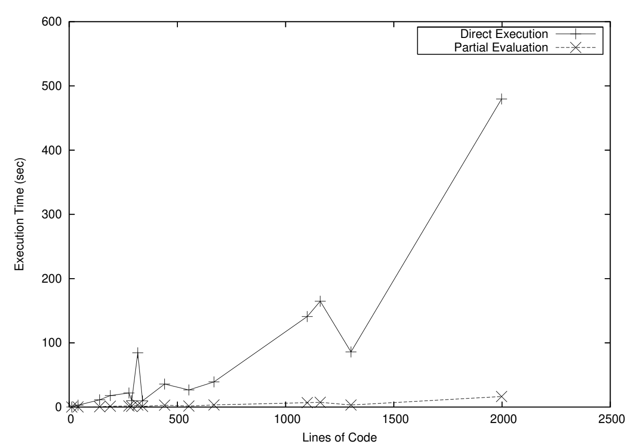

Figure 7: This figure shows how the performance of the Java sign analysis code scales with code size when directly executed versus partially evaluated. Constraint solving time was negligible in all cases, and is included in the partial evaluation numbers.

to directly execute the Prolog code with the time taken to partially evaluate the code and then solve the resulting constraints. As might be expected, direct execution scales very poorly to large programs. In contrast, our two-phase approach based on partial evaluation provides up to two orders of magnitude of performance improvement over direct execution.

4.2 Race Condition Checking for Java

As a second application, we re-implemented the `rccjava` type system [4] using this approach. The `rccjava` type system extends Java's type systems with additional checks to ensure that every well-typed program is free of race conditions. This extended type system requires additional type annotations that specify, for example, which a lock protects each field in the program, and which locks should be held on entry to each method.

The original implementation of this system [4] consists of approximately 12,000 lines of Java code (excluding the Javafe code shared with our current translator) and required three person-months of development effort. This implementation is focused on type checking. A subsequent separate implementation effort tackled type inference by reducing it to propositional satisfiability [5].

We have begun a re-implementation of this type system as a more significant scalability test for our approach. Currently, type checking operates smoothly. So far, the effort has taken approximately 0.25 person-months, and yielded about 300 lines of Prolog code. Our current partial evaluator cannot yet handle all of the Prolog features used in this type system. We are working on appropriately extending our partial evaluator to handle this type system. We also intend to explore type inference for atomicity [7] using our approach.

5. BEYOND PARTIAL EVALUATION

The approach described so far suffers from a performance overhead, in that partial evaluation involves essentially interpreting the type checking code, which is slower than direct execution of a hand-coded constraint generator.

```
base_subtype(B, B).

base_subtype(int, real).

subtype_F2(base(B1), base(B2), base_subtype(B1, B2)).

subtype_F2(func(A1, B1), func(A2, B2), C):-
    subtype_F2(A2, A1, C1),
    subtype_F2(B1, B2, C2),
    C = (C1, C2).

tc(_, int(_), base(int)).

tc(_, real(_), base(real)).

tc_F2(B, var(X), T, true):-
    member(bind(X, T), B).

tc_F2(B, apply(E1, E2), T, C):-
    tc_F2(B, E1, func(T1, T), C1),
    tc_F2(B, E2, T2, C2),
    subtype_F2(T2, T1, C3),
    C = (C1, C2, C3).

tc_F2(B, lambda(X, T1, E), func(T1, T2), C):-
    tc_F2([bind(X, T1)|B], E, T2, C).

tc_F2(B, if(E1, E2, E3), T, C):-
    tc_F2(B, E1, base(int), C1),
    tc_F2(B, E2, T2, C2),
    tc_F2(B, E3, T3, C3),
    subtype_F2(T2, T, C4),
    subtype_F2(T3, T, C5),
    C = (C1, C2, C3, C4, C5).

tc_F2(B, op(_, E1, E2), base(B3), C):-
    tc_F2(B, E1, base(B1), C1),
    tc_F2(B, E2, base(B2), C2),
    subtype_F2(base(B1), base(B3), C3),
    subtype_F2(base(B2), base(B3), C4),
    C = (C1, C2, C3, C4).
```

Figure 8: The result of applying the staging translation to the λ_{sub} type checker from Figure 3 (with the `base_subtype` predicate delayed), after α-renaming and peephole optimization.

We can address this interpretation overhead by using the second Futamura projection [10], which is often referred to as a *generating extension*. That is, we partially evaluate the application of the partial evaluator to the analysis code. The residual program is then a constraint generator that executes without any partial evaluation overhead. In more detail, if:

$$\texttt{peval(tc(}E\texttt{), }C\texttt{).}$$

partially evaluates the type checking code `tc` over the expression E, to yield a residual constraint C over type variables in E, then the goal:

$$\texttt{peval(peval(tc(}E\texttt{), }C\texttt{), }R\texttt{)}$$

yields a Prolog program R (over E and C) that provides a more efficient implementation of `peval(tc(`E`),` C`)`, in which the work of partial evaluation has already been performed.

Developing a partial evaluator that is self-applicable in this manner turns out to be quite difficult [2]. We can avoid this difficulty by instead writing a *staging transaction* that directly translates the goal $\texttt{tc}(E)$ into the desired second Futamura projection

$$R \equiv \texttt{tc\_F2}(E, C),$$

which takes as input an expression E and returns a conjunction C of delayed constraints for E.

This staging translation involves extending each user-defined predicate P with an output parameter C that contains a conjunction of delayed constraints. These delayed constraints include

1. any delayed constraint called directly from P, and
2. the delayed constraints returned from any non-delayed, user-defined predicate called from P.

Appendix B presents the code to perform this staging translation. It takes as input a program (as a list of clauses) and returns the second Futamura projection of that program (again, as a list of clauses). Although quite simple, this code is adequate for the type systems presented in this paper, and could easily be extended to support additional Prolog constructs.

Figure 8 shows the result of applying this staging translation to the λ_{sub} type checker from Figure 3 (with the `base_subtype` predicate delayed). The result is essentially identical to a hand-coded constraint generator, but avoids the cost of developing, debugging, and verifying the constraint generating code by hand.

6. RELATED WORK

A few papers have previously noted and tried to exploit the clear connections between type inference and type checking. Secher and Sørensen note that type checking is generalizable to type inference [20]. However, they base their analysis on the use of a deterministic, functional language to express type systems, which makes partial evaluation significantly harder. Our use of Prolog leads to code that closely matches the type rules, and enables straightforward partial evaluation.

Lu and King [16] also mention the connection between type inference and type checking. They note that, to quote the paper title, "backward type inference generalises type checking". However, they restrict their focus to the analysis of logic programs, within a specific context.

Frühwirth uses partial evaluation to perform type inference for logic programs [9], and proposes the use of logic programs themselves as types [8]. Like Lu and King, Frühwirth focuses specifically on types for logic programs.

A number of other researchers have developed systems that serve as ideal target constraint languages. Whaley and Lam developed a system called `bddbddb`, aimed at efficiently evaluating Datalog programs using BDDs [24]. It was designed for the purpose of program analysis, and would serve as an ideal evaluation tool for our larger Datalog programs.

BANE [1] and its successor, BANSHEE [13], by Aiken and Kodumal, provide platforms for developing type- and set-constraint-based program analyses. These systems focus more on solving constraints than generating them, and could be useful as target constraint languages for our approach.

Venkatesh and Fischer's paper on SPARE [23] describes a general-purpose program analysis environment and discusses the sign analysis problem that we use as an example. It might be possible to write our analyses within SPARE, instead of Prolog, but we do not have access to a SPARE system, and the resulting code probably would not fit the type rules as closely as Prolog code can.

Crew, from Microsoft Research, developed a language called ASTLOG [3] which has a Prolog-like syntax and built-in support for extracting AST information from C programs. It was intended mostly as tool for simple search problems, however, and is not ideal for more complex program analyses such as type inference.

LIX [2] is a self-applicative partial evaluator for Prolog that provided some ideas about how to design our own partial evaluator, but LIX does not provide the ability to delay predicates in the way we need. We plan to develop more powerful partial evaluation strategies in the future, however, that may benefit from the more advanced features of LIX, including its self-applicability. In addition to LIX, many other partial evaluators exist, including Mixtus [21], Ecce [15], and ProMiX [14].

7. CONCLUSION AND FUTURE WORK

Our work is motivated by the central role type rules play in the development of the various algorithms related to a type system, from checking to inference, which all proceed by either verifying or solving the same set of constraints. While the actual algorithm used for type checking and type inference vary greatly, we have demonstrated that a single natural encoding of the original type rules can be used to derive these various analysis algorithms. Furthermore, while our approach applies most obviously to type systems, *per se*, it is also useful for program analysis problems that may not be considered strictly type systems but have a similar structure.

Prolog clauses provide a natural and convenient way to encode these type rules. This code can then directly execute with any combination of specified and unspecified types and, if computationally possible (with a finite proof tree), infer bindings for the unspecified types. If many types are left unspecified, however, standard Prolog search strategies result in inefficient execution. If, instead, we identify the operations within the type system that depend on the bindings of the type variables, and use partial evaluation to delay those operations, we can automatically derive a residual constraint (in the form of a Prolog goal) over the type variables. A variety of external solvers can then efficiently solve the resulting constraint system.

Furthermore, the second Futamura projection [10] can be used to remove the partial evaluation overhead from the constraint-generating phase, resulting in an automatically-generated constraint generator that is essentially identical to a generator that would be written by hand.

Finally, while Prolog provides an almost satisfactory language for the description of type systems and similar program analyses, it has some shortcomings. In particular, it lacks a static type system of its own to detect errors at compile time, and is somewhat limited in its facilities for abstraction. Moving to a strongly-typed logic language, such as Mercury [22], might alleviate some of the implementation difficulties. Alternatively, a language such as Twelf [18] or Delphin [19] might be applicable, as they were both designed for the analysis of programming languages, logics, and proofs.

Acknowledgments This work was partly supported by the National Science Foundation under Grant CCR-03411797 and by faculty research funds granted by the University of California at Santa Cruz.

8. REFERENCES

[1] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the 2nd Annual Workshop on Types in Compilation,TIC'98*, July 1998.

[2] S.-J. Craig and M. Leuschel. LIX: an effective self-applicable partial evaluator for prolog. In *Proceedings of the 7th International Symposium on Functional and Logic Programming, FLOPS'04*, Apr. 2004.

[3] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Oct. 1997.

[4] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'00*, June 2000.

[5] C. Flanagan and S. N. Freund. Type inference against races. In *Proceedings of the 2004 Static Analysis Symposium, SAS'04*, Aug. 2004.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'02*, May 2002.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'03*, June 2003.

[8] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 300–309, 1991.

[9] T. W. Frühwirth. Type inference by program transformation and partial evaluation. In *Proceedings of the Workshop on Meta-Programming in Logic (META'88)*, pages 263–282, 1988.

[10] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[11] S. Horowitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115, 1995.

[12] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java^{TM}Language Specification, Second Edition.* Addison-Wesley, 2000.

[13] J. Kodumal. *Banshee, a toolkit for building constraint-based analyses.* PhD thesis, University of California at Berkeley, 2002.

[14] A. Lakhotia and L. Sterling. ProMiX: a Prolog partial evaluation system. pages 137–179.

[15] M. Leuschel. The ECCE partial deduction system. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, 1997.

[16] L. Lu and A. King. Backward type inference generalises type checking. In *Proceedings of the 9th International Symposium on Static Analysis, SAS'02*, Aug. 2002.

[17] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference, DAC'01*, June 2001.

[18] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction, CADE-16*, July 1999.

[19] C. Schürmann. Towards practical functional programming with logical frameworks, July 2003.

[20] J. P. Secher and M. H. Sørensen. From checking to inference via driving and DAG grammars. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, PEPM'02*, pages 41–51. ACM Press, Jan. 2002.

[21] D. Shalin. Mixtus: an automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1994.

[22] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, Feb. 1995.

[23] G. A. Venkatesh and C. N. Fischer. SPARE: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 18(4), Apr. 1992.

[24] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'04*, June 2004.

APPENDIX

A. PARTIAL EVALUATOR

```
:- multifile delay/1.
:- multifile partdelay/2.
:- dynamic saved/2.

peval(P, R) :- delay(P), !, R = P.

peval((P1, P2), (R1,R2)) :-
    !, peval(P1, R1), peval(P2, R2).

peval(P, R) :-
    P =.. [Pred|_],
    member(Pred, [member,=,true,fail,<,>]),
    !, P, R = true.

peval(P, R) :-
    partdelay(P, _), saved(P, _),
    !, trim_goal(P, R).

peval(P, R) :-
    partdelay(P, _),
    clause(P, Q), peval(Q, R1),
    trim_goal(P, P1),
    assert(saved(P1, R1)), R = P1.

peval(P, R) :- clause(P, Q), peval(Q, R).

/* Deconstruct a predicate, remove unwanted
 * parameters, and reconstruct it.
*/

trim_goal(P, R) :-
    P =.. [Pred|Args],
    partdelay(P, S),
    crop_list(Args, S, NArgs),
    R =.. [Pred|NArgs].

crop_list([], [], []).

crop_list([IH|IT], [n|ST], [IH|Rest]) :-
    crop_list(IT, ST, Rest).

crop_list([_|IT], [y|ST], Rest) :-
    crop_list(IT, ST, Rest).

/* Write out the Datalog program resulting
 * from partial delays.
*/

write_saved :-
    forall(saved(P, R),
        (write(P), write(' :- '),
        write(R), write('.'), nl)).
```

B. STAGING TRANSFORMATION

```
stage_program([], []).

stage_program([Clause|Tail], [NewClause|NewTail]) :-
    stage_clause(Clause, NewClause),
    stage_program(Tail, NewTail).

stage_clause(P :- B, P :- B) :- delay(P), !.

stage_clause(P :- B, NewP :- (NewB,C=Cs)) :-
    !,
    P =.. [Pred|Args],
    stage_term(B, NewB, Cs),
    append(Args, [C], NewArgs),
    atom_concat(Pred, '_F2', NewPred),
    NewP =.. [NewPred|NewArgs].

stage_clause(C, C).

stage_term((A, B), (NewA, NewB), (AC, BC)) :-
    stage_term(A, NewA, AC),
    stage_term(B, NewB, BC).

stage_term(A, true, A) :- delay(A), !.

stage_term(A, A, true) :-
    A =.. [Pred|_],
    member(Pred, [=,member,<,>,true,fail]).

stage_term(A, NewA, Cs) :-
    A =.. [Pred|Args],
    atom_concat(Pred, '_F2', NewPred),
    append(Args, [Cs], NewArgs),
    NewA =.. [NewPred|NewArgs].
```