Inference of Static Semantics for Incomplete C Programs

LEANDRO T. C. MELO, UFMG, Brazil
RODRIGO G. RIBEIRO, UFOP, Brazil
MARCUS R. DE ARAÚJO, UFMG, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Incomplete source code naturally emerges in software development: during the design phase, while evolving, testing and analyzing programs. Therefore, the ability to understand partial programs is a valuable asset. However, this problem is still unsolved in the C programming language. Difficulties stem from the fact that parsing C requires, not only syntax, but also semantic information. Furthermore, inferring types so that they respect C's type system is a challenging task. In this paper we present a technique that lets us solve these problems. We provide a unification-based type inference capable of dealing with C intricacies. The ideas we present let us reconstruct partial C programs into complete well-typed ones. Such program reconstruction has several applications: enabling static analysis tools in scenarios where software components may be absent; improving static analysis tools that do not rely on build-specifications; allowing stub-generation and testing tools to work on snippets; and assisting programmers on the extraction of reusable data-structures out of the program parts that use them. Our evaluation is performed on source code from a variety of C libraries such as GNU's Coreutils, GNULib, GNOME's GLib, and GDSL; on implementations from Sedgewick's books; and on snippets from popular open-source projects like CPython, FreeBSD, and Git.

CCS Concepts: • **Theory of computation → Type structures**; **Type theory**; **Program analysis**; *Parsing*; • **Software and its engineering → Source code generation**; *Constraints*; *Syntax*; *Parsers*;

Additional Key Words and Phrases: Partial Programs, Parsing, Type Inference, C Language

ACM Reference Format:
Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. 2018. Inference of Static Semantics for Incomplete C Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 29 (January 2018), 28 pages. https://doi.org/10.1145/3158117

1 INTRODUCTION

Incomplete source-code appears in a variety of scenarios: during inception of a program, within an editor or IDE; in cross-platform development, when source portions are unavailable due to incompatibility; in the form of patches submitted to code reviewing; as snippets contained in reports from bug-trackers. Therefore, due to the ubiquitousness of partial programs, the ability to work with incomplete sources is a desirable asset.

Testimony of this importance is the fact that the programming languages community has gone to great lengths to design tools that can deal with partial programs [Chugh et al. 2009; Dagenais and Hendren 2008; Godefroid 2014; Knapen et al. 1999; Koppler 1997; Perelman et al. 2012]. However,

Authors' addresses: Leandro T. C. Melo, UFMG, Belo Horizonte, Minas Gerais, Brazil, ltcmelo@dcc.ufmg.br; Rodrigo G. Ribeiro, UFOP, Ouro Preto, Minas Gerais, Brazil, rodrigo@decsi.ufop.br; Marcus R. de Araújo, UFMG, Belo Horizonte, Minas Gerais, Brazil, maroar@dcc.ufmg.br; Fernando Magno Quintão Pereira, UFMG, Belo Horizonte, Minas Gerais, Brazil, fernando@dcc.ufmg.br.

2475-1421/2018/1-ART29
https://doi.org/10.1145/3158117

in the realm of the C programming language, this is still an unsolved problem. This difficulty stems mainly from two reasons: (i) the tight coupling between syntactic and semantic analysis when parsing C code; (ii) the lack of a type inference engine for C, granted the challenges involved in building one. This paper describes solutions to these problems.

In this work, we present what, to the best of our knowledge, is the first technique thats allows the compilation of incomplete C sources. Our goal is to *reconstruct* a partial program $\mathcal{P}$ as a new program $\mathcal{P}'$ that preserves every syntactic construction of $\mathcal{P}$, and contains any type declaration missing from $\mathcal{P}$, so that the new program $\mathcal{P}'$ is well-typed. As demonstrated in Section 5, such successful reconstruction enables static analysis tools to work, where otherwise not viable; improves precision of "zero setup" based static analysis tools; supports testing and stub-generation; and serves as a general code completer for C programmers.

The Contributions of This Work. During the course of dealing with incomplete C sources, we architect solutions to the following problems:

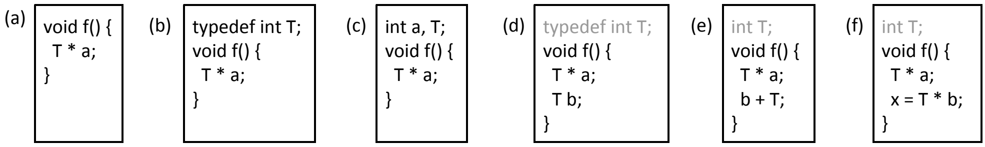
- Parsing C requires semantic information to handle ambiguous syntax, but this information might be missing. In Section 3.1 we formalize a technique that deals with this problem: decisions are postponed until further syntax can be extracted from the syntax available in the partial program.
- C accepts liberal conversions, for instance, between a pointer type and an integer type (i.e. the *null pointer constant*, 0). But those two types are not syntactically interchangeable, thus they are not unifiable. In Section 3.2 we explain how to find the specific type of a variable, by defining a lattice of pre-types to solve this problem.
- In C, implicit conversions of qualified types (e.g const and volatile) are asymmetric. This prevents standard type inference techniques, which rely on type equivalences, to work with C. In Section 3.3 we discuss a strategy to model pointer relations through subtyping and how we use unification to simultaneously solve constraints in the form of type equivalence and type inequality.
- An incomplete source code might not contain enough usages of its variables and its functions so that their type can be inferred. In Section 3.4 we describe how our type inference engine works under such constraint. Unsolved types can be safely instantiated, without interfering with solved type variables.

To demonstrate the ideas advocated in this paper, we have materialized them into a tool called PsycheC which produces a C header containing declarations that are absent from the partial program it receives as input. The #inclusion of this header in the incomplete source characterizes a reconstructed program that compiles successfully. Notice that, in the context of C, referring to "compilation" as the entire pipeline of (a) preprocessing, (b) compiling, (c) assembling, and (d) linking is a common abuse of terminology. Our work is specifically targeted at compilation, as in item (b). We do not generate definitions for missing functions, so the program reconstructed by PsycheC may not link - stub-generation tools exist [Cadar et al. 2008; Godefroid et al. 2005; Tillmann and De Halleux 2008; Williams et al. 2005], but are beyond the scope of this paper. A description of our implementation appears in Section 4.

2 CHALLENGES

To produce a well-typed program out of an incomplete C source we have to circumvent a number of challenges. In this section, we provide examples of such challenges, and in Section 3 we show how we solve them. Inferring types that satisfy C's type system involves other challenges not mentioned in this section. Those are discussed throughout the text. We start with Challenge 1, which concerns the parsing of a partial C program.

Challenge 1. Determine the syntactic nature of user-defined names in a programming language that relies on semantic information to guide parsing, when declarations are missing.

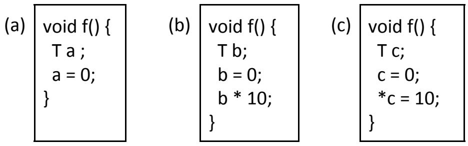


(a)
```
void f() {
  T * a;
}
```
(b)
```
typedef int T;
void f() {
  T * a;
}
```
(c)
```
int a, T;
void f() {
  T * a;
}
```
(d)
```
typedef int T;
void f() {
  T * a;
  T b;
}
```
(e)
```
int T;
void f() {
  T * a;
  b + T;
}
```
(f)
```
int T;
void f() {
  T * a;
  x = T * b;
}
```

Fig. 1. (a) Is T the name of a variable or of a type? (b) Complete program where T is a type. (c) Complete program where T is a variable. (d) Syntax that lets us conclude T is a type. (e-f) Syntax that lets us conclude T is a variable.

Figure 1 illustrates this challenge. The program in part (a) does not contain enough information to determine the syntactic nature of T. Depending on missing elements, T can denote a type, as in Figure 1 (b), or a variable, as in Figure 1 (c). As we show in Section 3.1, to determine the meaning of T, we postpone parsing decisions until we have seen more of the program. For instance, extra syntax lets us infer that T is a type in Figure 1 (d); or a variable in Figures 1 (e and f).

Challenge 2. Distinguish between non-unifiable types that are mutually exchangeable.

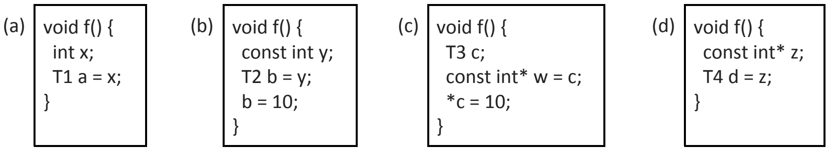


(a)
```
void f() {
  T a ;
  a = 0;
}
```
(b)
```
void f() {
  T b;
  b = 0;
  b * 10;
}
```
(c)
```
void f() {
  T c;
  c = 0;
  *c = 10;
}
```

Fig. 2. (a) Is T numeric or a pointer? (b) Program where T is numeric. (c) Program where T is pointer.

The program in Figure 2 (a) is not syntactically ambiguous: T must be a type. However, this program is semantically ambiguous: T can be int, int*, int**, float, float*, etc. As we explain in Section 3.2, we define a lattice of pre-types that helps us find *the most general type* of T in those examples. This lattice lets us promote T to *numeric* in Figure 2 (b) and to *pointer* in Figure 2 (c). To bind names to points in this lattice, we rely on further program syntax. For instance, in Figure 2 (b) we know that b must be an arithmetic type[1], because pointers cannot be used in multiplication [ISO-Standard 2011]{§6.5.5}. On the other hand, Figure 2 (c) declares a pointer: the syntax of C does not permit dereferencing arithmetic types, as we observe in line 4.

Challenge 3. Account for unidirectional type equivalences represented by assignments.



(a)
```
void f() {
  int x;
  T1 a = x;
}
```
(b)
```
void f() {
  const int y;
  T2 b = y;
  b = 10;
}
```
(c)
```
void f() {
  T3 c;
  const int* w = c;
  *c = 10;
}
```
(d)
```
void f() {
  const int* z;
  T4 d = z;
}
```

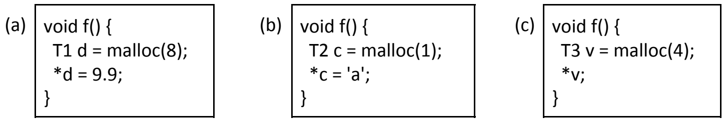
Fig. 3. (a) Is T1 int or const int? (b) Despite y being const int, T2 cannot be const int. (c) Despite w being const int*, T3 cannot be const int*. (d) Program where T4 must be const int*.

This challenge exists due to the way *type qualifiers* [ISO-Standard 2011]{§6.7.3} work in C. We use const throughout the text, but the described behaviour applies to volatile as well. In Figure 3

[1] The C standard refers to integer and floating-point types collectively as arithmetic types [ISO-Standard 2011]{§6.2.5.21}.

(a), T1 can be either int or const int. The latter is possible because creating a *constant* variable out of a non-constant one is allowed. But, in Figure 3 (b) T2 must be int: T2 as a const int would lead to an invalid program, since reassigning a constant, as in b = 10, is not permitted. A harder challenge appears in Figure 3 (c), where the assignment involves pointers. Despite w being const int*, it may be assigned to a more strict pointer. In fact, the expression *c = 10 indicates that c cannot be a constant pointer and, therefore, the only correct solution is T3 as an int*. However, in Figure 3 (c), since the assignment is *from* a constant pointer, T4 cannot be int*. It must really be const int*. Due to the aforementioned asymmetries, traditional unification, which relies on type equivalences, cannot be used in C. In Section 3.3, we discuss how we employ subtyping to solve this problem and a novel approach to incorporated it in unification.

Challenge 4. Generate types for variables whose nature is not sufficiently restricted by syntax.



(a)

```
void f() {
  T1 d = malloc(8);
  *d = 9.9;
}
```

(b)

```
void f() {
  T2 c = malloc(1);
  *c = 'a';
}
```

(c)

```
void f() {
  T3 v = malloc(4);
  *v;
}
```

Fig. 4. What are the types T1, T2 and T3, considering that malloc's return type is void*?

Figure 4 illustrates this challenge. The return type of malloc is void*. In programming language parlance, void* is a *top type* among pointers, meaning that we can unify it with any other pointer type. However, there is no actual value whose type is only void*. And yet, we need to instantiate it to produce a well-typed program for the incomplete source in Figure 4 (c). Further syntax in Figures 4 (a-b) lets us conclude that T1 and T2 are arithmetic types; on the other hand, in Figure 4 (c) we do not have this information. In Section 3.4 we define the notion of "orphan" variables, which we always can safely instantiate. In Figure 4 (c), v is a pointer to an orphan.

3 FROM INCOMPLETE SOURCES TO WELL-TYPED PROGRAMS

To produce a well-typed program out of an incomplete C source-code $\mathcal{P}_{par}$, we proceed in four steps. First, we parse $\mathcal{P}_{par}$ (Section 3.1) to obtain an abstract syntax tree (AST). Second, we traverse this AST to generate constraints (Section 3.2). Third, we use an unification-based algorithm to solve these constraints (Section 3.4). Finally, solved constraints let us infer missing types for $\mathcal{P}_{par}$ (Section 3.4). We describe these steps in the rest of this section. During this discussion, we assume that $\mathcal{P}_{par}$ is derived from a syntactically valid program $\mathcal{P}_{ori}$. During our presentation, three core languages are developed: μA, μB, and μC. They contain only enough syntax to illustrate the essence of our ideas. PsycheC accepts actual C. Section 4 describes information about the C standard version that our tool can deal with.

3.1 Challenge 1 – Parsing Ambiguous Syntax

The C programming language uses a symbol table to guide parsing. The contents of this symbol table determine which production will be matched, depending on whether a name designates either a variable or a type. For instance, the term x*y; can denote either the product of variables x and y, or the declaration of variable y as a pointer to type x. Ambiguities do not happen in a complete C program because declarations must precede uses of identifiers [ISO-Standard 2011]. However, incomplete programs may not contain all type declarations, thus making it impractical for the parser to rely on such semantic information. A known approach to gather information from incomplete programs is *fuzzy parsing* [Koppler 1997], but a fuzzy parser does not recognise the entire language, only certain constructs of interest. Knapen *et al* [Knapen et al. 1999] presents a technique to deal with C++ ambiguities that resembles fuzzy parsing. They point out eight ambiguous syntactic constructions, out of those, three exist in C as well. Figure 5 shows them. In the rest of this section, we explain how we solve ambiguities through a technique similar to Knapen's, but under formalised guarantees on the produced AST.

Function call *or* variable declaration	a(b);
Coercion of unary expression *or* binary expression	(a)*b; (a)-b;
Pointer declaration *or* multiplication	a*b;

Fig. 5. Ambiguities due to missing declarations.

Properties of Partial-Program ASTs. The key idea to deal with missing declarations is to postpone ambiguity resolution until we have enough information to solve it. To explain this idea, we shall use the grammar in Figure 6. This is the minimum setup that lets us build the ambiguous term "T * a". The other cases in Figure 5 are similar. We borrow the notation of Prolog's *logical grammars* [Sterling 1994, Ch.19], e.g., the grammar in Figure 6 is an executable Prolog program. We shall call the language defined by that grammar μA. A program is a list of *terms* separated by semicolon (;). Terms can be type declarations (*Td*), variable declarations (*Vd*) or expressions (E). The non-terminal that denotes programs has four attributes, e.g.: $P(T_0, V_0, T_1, V_1)$. Following common parsing jargon, the first two are *inherited*, and the others are *synthesized*. Thus, the production P receives two attributes, T_0 and V_0, and, if it successfully consumes a string, then it produces the attributes T_1 and V_1. We let T denote sets of names used as types, and V denote sets of names used as variables. Terms, which we denote by S, use the same four attributes. A type declaration $Td(x)$ succeeds if x is found to be the name of a new type. Only the type int is concrete in μA; hence, new types are alias of int. A variable declaration $Vd(T, x)$ succeeds if x can be proven to be the name of a variable whose type is present in the set of types T. Parsing an expression such as $E(V)$ succeeds if all the variables that this expression uses are present in the set of names V.

Definition 3.1 (Valid program). We say that $\mathcal{P}$ is a valid program if $\mathcal{P}$ is a list of terms, e.g., $(Td; \mid Vd; \mid E;)*$ that can be consumed by the production rule $P(\emptyset, \emptyset, T, V)$ in Figure 6. In this case, we say that T is the set of *type names*, and V is the set of *variable names* of $\mathcal{P}$.

Lemma 3.2. *The following properties are true about a valid program $\mathcal{P}$ produced by $P(\emptyset, \emptyset, T, V)$. We let $a \mathbin{+\!\!+} b$ denote the concatenation of the lists a and b:*

(1) $T \cap V = \emptyset$

(2) If $\mathcal{P} = \mathcal{P}_1 \mathbin{+\!\!+} a\, b; \mathbin{+\!\!+}\ \mathcal{P}_2$, then: (i) there exists x, such that (typedef x a;) $\in \mathcal{P}_1$*, and (ii)* $x \in T$

$$\begin{array}{rcll}
P(T,V,T”,V”) & ::=_1 & S(T,V,T',V');P(T',V',T”,V”); & \\
P(T,V,T',V') & ::=_2 & S(T,V,T',V'); & \\
S(T,V,T \cup \{x\},V) & ::=_3 & Td(x) & \text{if } x \notin V \\
S(T,V,T,V \cup \{x\}) & ::=_4 & Vd(T,x) & \text{if } x \notin T \\
S(T,V,T,V) & ::=_5 & E(V) & \\
Td(x) & ::=_6 & \text{typedef int } x & \\
Vd(T,y) & ::=_7 & x\ y & \text{if } x \in T \\
Vd(T,y) & ::=_8 & x * y & \text{if } x \in T \\
E(V) & ::=_9 & x + y & \text{if } x \in V \wedge y \in V \\
E(V) & ::=_{10} & x * y & \text{if } x \in V \wedge y \in V
\end{array}$$

Fig. 6. μA – definition of a minimalistic C program. The subscript s in the production symbol, e.g., $::=_s$ helps us writing the proofs (available as supplementary material).

$$
\begin{array}{lll}
P_\beta(T,V,T'',V'') & ::=_a & S_\beta(T,V,T',V'); P_\beta(T',V',T'',V''); \\
P_\beta(T,V,T',V') & ::=_b & S_\beta(T,V,T',V'); \\
S_\beta(T,V,T,V \cup \{x\}) & ::=_c & TV_\beta(x) \\
S_\beta(T,V,T,V \cup \{x,y\}) & ::=_d & E_\beta(x,y) \\
S_\beta(T,V,T \cup \{x\},V) & ::=_e & Td_\beta(x) \\
S_\beta(T,V,T \cup \{x\},V \cup \{y\}) & ::=_f & Vd_\beta(x,y) \\
TV_\beta(y) & ::=_g & x * y \\
Td_\beta(x) & ::=_h & \text{typedef int } x \\
Vd_\beta(x,y) & ::=_i & x\ y \\
E_\beta(x,y) & ::=_j & x + y
\end{array}
$$

Fig. 7. Grammar that we use to disambiguate programs. We refer to this new language as μB. Its grammar is similar to that of μA, but with a new non-terminal *TV*, which, despite deriving syntax [x $*$ y], lets us postpone the decision of whether x is a variable or a type.

(3) *If $\mathcal{P} = \mathcal{P}_1 +\!\!+\ a + b; +\!\!+\ \mathcal{P}_2$, then: (i) there exists x, such that $(x\ a;) \in \mathcal{P}_1$, (ii) there exists y, such that $(y\ b;) \in \mathcal{P}_1$, (iii) $\{x,y\} \subseteq T$ and (iv) $\{a,b\} \subseteq V$*

Example 3.3. Program $\mathcal{P}_1$ = [typedef int a; a a;] is not valid, because name a is used both as variable and as type. Program $\mathcal{P}_2$ = [a b;] is not valid, because the name a is used as a type, but has not been defined before. Program $\mathcal{P}_3$ = [int a; a + b;] is not valid because the name b is used as a variable, but has not been defined before.

Definition 3.4 (Partial Program). Let $\mathcal{P}_{ori}$ be a list of terms that denotes a valid program. We obtain a partial program $\mathcal{P}_{par}$ by eliminating any number of terms from $\mathcal{P}_{ori}$.

Example 3.5. The program $\mathcal{P}$ = [typedef int a; a b; a $*$ c;] is valid according to definition 3.1. There exist eight possible partial programs that we can produce out of $\mathcal{P}$. A few of them are shown below: $\mathcal{P}_1$ = [a b; a $*$ c;]. $\mathcal{P}_2$ = [typedef int a; a $*$ c;]. $\mathcal{P}_3$ = [a b; a $*$ c;].

As we notice from Example 3.5, some partial programs are not valid. Some invalid μA programs correspond to ambiguous C codes. For instance, the program [a $*$ b;] is ambiguous, because we do not know if it corresponds to a multiplication between a and b, or the declaration of b as a pointer of type a. However, there are partial programs that provide us with enough information to disambiguate them. As an example, the program [a $*$ b; a c;] can only be parsed as two declarations. In this case, the second term, e.g., [a c;], lets us infer that the name a is a type, not a variable. Similarly, the program [a $*$ b; a + c;] can only denote two expression statements, for the second term lets us infer that a is the name of a variable. Based on this observation, we use the logical grammar in Figure 7 to disambiguate partial programs.

The grammar in Figure 7 uses a new non-terminal *TV* to disambiguate names. To this effect, upon finding a term such as [a $*$ b], we postpone the classification of a as either type or variable. This name, a, will only be marked as a type if the partial program contains either a term such as [typedef int a], or a term such as [a b]. In the latter case, the name a is being used as the type of a variable b. Similarly, we mark a as a variable if the partial program contains either a term such as [x a] (declaration of variable a), or an expression such as [x + a], whose syntax only admits the name a being a variable.

Definition 3.6 (Successful Disambiguation). Let $\mathcal{P}_{par}$ be a partial program. We say that we can successfully disambiguate $\mathcal{P}_{par}$ if the production rule $P_\beta(\emptyset, \emptyset, T, V)$ succeeds on $\mathcal{P}_{par}$, and every name in $\mathcal{P}_{par}$ is in T or in V. In this case, we say that $\mathcal{P}_{par}$ is an *unambiguous program.*

Notice that the parsing in Figure 7 succeeds in some programs that are still ambiguous, e.g., [int x; a $*$ x;]. In this program we cannot tell the nature of a. Thus, for a successful disambiguation we require every name in the program to be classified as either a type or a variable. Nevertheless, we can prove several properties of partial programs, even if they are still ambiguous, as we state in Theorem 3.7.

Theorem 3.7. *Let $\mathcal{P}_{par}$ be a partial program derived from $\mathcal{P}_{ori}$. If $P_\beta(\emptyset, \emptyset, T_\beta, V_\beta)$ succeeds on $\mathcal{P}_{par}$, and $P(\emptyset, \emptyset, T, V)$ succeeds on $\mathcal{P}_{ori}$, then the following properties hold:*

(1) If $x \in T_\beta$, then $x \in T$
(2) If $x \in V_\beta$, then $x \in V$

Theorem 3.7 gives us a simple corollary, stated below:

Corollary 3.8. *Let $\mathcal{P}_{par}$ be a partial program. If $P_\beta(\emptyset, \emptyset, T_\beta, V_\beta)$ succeeds on $\mathcal{P}_{par}$, then $T_\beta \cap V_\beta = \emptyset$.*

3.2 Challenge 2 – Dealing with Ambiguous Types

C allows a programmer to define new types (via the enum, struct and union constructs), and to create aliases to existing types via the typedef construct. Missing type declarations impose an obstacle to the compilation of incomplete C programs. To reconstruct these declarations, we perform type inference in two stages: constraint generation and constraint solving, following common practice for type inference in Haskell and ML [Rémy 2013, Ch.5]. However, in our case, standard type inference might not be enough, due to ambiguities. The same syntax in C can denote different types. Figure 2 already shows an example of such ambiguity, because the constant 0 can be assigned either to a numeric type or a pointer. There are other examples that lead to similar problems. For instance, the expression {1, 2, 3, 4} can denote different kinds of aggregate types[2]: an array of integers (int[]) or a struct with four integer fields (struct T {int a, b, c, d}) [ISO-Standard 2011]{§6.7.8}.

To solve type ambiguities, we use global information to determine the type of local constructs. To this end, we build a lattice of "pre-types". Pre-types are not standard C types; rather, they work as placeholders for them. Every type variable that we shall create during constraint generation is bound to a pre-type. During constraint generation we find syntax that lets us move unresolved type variables up this lattice, until reaching a fixed-point. We use one lattice to solve the ambiguity between pointer and numeric types seen in Figure 2, and another to solve ambiguities between aggregate types. These lattices have different structure, but they are built using the same ideas. Thus, we focus on the former in the rest of this section. To explain our resolution strategy, we replace μB with μC, a language that contains syntax to deal with arithmetic and pointer types[2]. Figure 8 defines the syntax of μC. Notice that μC's syntax is not a superset of μB's; for instance, the latter contains a typedef construct, which is absent in the former.

The Semantics of Constraints. Following Nielson *et al.* [Nielson et al. 2005, Ch.3] or Pottier and Rémy [Pottier and Rémy 2005], we determine the properties of an acceptable set of constraints before we show how to generate them. The semantics of constraints, outlined in Figures 9 and 10, is defined as the judgment ϕ; ψ; $\Theta \models K$, where each mapping is defined as follows:

- ϕ maps type variables (α) to types (τ), e.g., $\phi(\alpha_1) = \text{const int}$ or $\phi(\alpha_2) = \{x : \text{char}, y : \text{char}\}$.
- ψ maps program variables to types, e.g., $\psi(x) = \text{int}^*$ or $\psi(y) = \mathbf{t}$.
- Θ maps types to types, e.g., $\Theta(\text{int}) = \alpha_1$, $\Theta(\text{int}) = \mathbf{t}$, or $\Theta(\mathbf{t}) = \text{u}^*$.

[2] The C standard refers to array and structures as aggregate types, and to arithmetic and pointers as scalar types [ISO-Standard 2011]{§6.2.5.21}.

$$
\begin{array}{lcll}
P & ::= & D\,P \mid D & ;\text{Program} \\
D & ::= & \tau\ f\ (\tau\ x)\ \{S\} & ;\text{Function} \\
S & ::= & S\,S & ;\text{Statements} \\
 & \mid & E; & ;\text{Expr-stmt.} \\
 & \mid & \tau\ x; & ;\text{Decl-stmt.}
\end{array}
\qquad
\begin{array}{lcll}
E & ::= & \ell & ;\text{Literal} \\
 & \mid & x & ;\text{Variable} \\
 & \mid & E\text{->}x & ;\text{Field access} \\
 & \mid & {}^*E & ;\text{Dereference} \\
 & \mid & \&E & ;\text{Address-of} \\
 & \mid & E = E & ;\text{Assignment} \\
 & \mid & E \oplus E & ;\text{Bin. op.}
\end{array}
\qquad
\begin{array}{lcll}
\tau & ::= & \mathbf{t} & ;\text{Concrete type} \\
 & \mid & \tau^* & ;\text{Pointer} \\
 & \mid & \mathsf{const}\ \tau & ;\text{Qualified type} \\
 & \mid & \{x_i : \tau_i\}^{i=1..n} & ;\text{Record} \\
 & \mid & \tau \to \tau & ;\text{Function type} \\
 & \mid & \alpha & ;\text{Type variable}
\end{array}
$$

Fig. 8. Syntax of μC

$$
\frac{\mathsf{conv}(\mathbf{t}_1, \mathbf{t}_2)}{\phi \vdash \mathbf{t}_1 \equiv \mathbf{t}_2}
\qquad
\frac{\phi \vdash \tau_1 \equiv \tau_2}{\phi \vdash \mathsf{const}\ \tau_1 \equiv \mathsf{const}\ \tau_2}
\qquad
\frac{\phi \vdash \tau_1 \equiv \tau_2}{\phi \vdash \tau_1^* \equiv \tau_2^*}
\qquad
\frac{\forall i, \phi \vdash \tau_{i1} \equiv \tau_{i2}}{\phi \vdash \{x_i : \tau_{i1}\}^{i=1..n} \equiv \{x_i : \tau_{i2}\}^{i=1..n}}
$$

$$
\frac{\phi(\alpha) = \tau}{\phi \vdash \alpha \equiv \tau}
\qquad
\frac{\phi(\alpha) = \tau}{\phi \vdash \tau \equiv \alpha}
\qquad
\frac{\phi \vdash \tau_1 \equiv \tau_2 \quad \phi \vdash \tau_1' \equiv \tau_2'}{\phi \vdash \tau_1 \to \tau_1' \equiv \tau_2 \to \tau_2'}
\qquad
\frac{\phi \vdash \tau_1 \equiv \tau_2}{\phi \vdash \tau_1 \leq \tau_2}
\qquad
\frac{\phi \vdash \tau_1 \equiv \tau_2}{\phi \vdash \tau_1^* \leq \mathsf{const}\ \tau_2^*}
$$

Fig. 9. Definition of type equivalence. We let $\mathsf{conv}(\mathbf{t}_1, \mathbf{t}_2)$ be true if $\mathbf{t}_1$ and $\mathbf{t}_2$ are mutually convertible concrete types, e.g., $\mathbf{t}_1 = \mathsf{int}$ and $\mathbf{t}_2 = \mathsf{char}$, or if they have the same name, e.g., $\mathbf{t}_1 = \mathbf{t}_2$.

$$
\phi; \psi; \Theta \models \top
\qquad
\frac{\exists \alpha. \psi(x) = \alpha \quad \phi(\alpha) = \tau}{\phi; \psi; \Theta \models \mathit{typeof}(x, \tau)}
\qquad
\frac{\phi \cup \{\alpha \mapsto \tau\}; \psi; \Theta \models K[\alpha \mapsto \tau]}{\phi; \psi; \Theta \models \exists \alpha. K}\ \forall \tau. \mathit{fv}(\tau) = \emptyset
$$

$$
\frac{\phi \vdash \tau_1 \leq \tau_2}{\phi; \psi; \Theta \models \tau_1 \equiv \tau_2}
\qquad
\frac{x : \tau_2 \in \mathit{fields}(\Theta(\tau)) \quad \tau_2 \leq \tau_1}{\phi; \psi; \Theta \models \mathit{has}(\tau, x : \tau_1)}
\qquad
\frac{\exists \alpha. \phi \cup \{\alpha \mapsto \tau\}; \psi \cup \{x \mapsto \alpha\}; \Theta \models K}{\phi; \psi; \Theta \models \mathit{def}\ x : \tau\ \mathit{in}\ K}
$$

$$
\frac{\phi; \psi; \Theta \models K_1 \quad \phi; \psi; \Theta \models K_2}{\phi; \psi; \Theta \models K_1 \wedge K_2}
\qquad
\frac{\phi; \psi; \Theta \models \Theta(\tau) = \phi(\alpha)}{\phi; \psi; \Theta \models \mathit{typedef}\ \tau\ \mathit{as}\ \alpha}
$$

Fig. 10. The semantics of constraints.

A constraint K is satisfiable if there exist ϕ, ψ, and Θ such that $\phi;\ \psi;\ \Theta \models K$ holds. Along this presentation, we shall use ℓ for literals; τ for types; and $\mathbf{t}$ for concrete types, e.g., builtins like int and char, or type names introduced by the programmer. We use = for assignments and $\oplus$ for the other binary operators (e.g.: +, ×, ||, etc); D for declarations; S for statements; and E for expressions. We let $\rho : \ell \to \tau$ be a function that maps literals to types, e.g., $\rho(1) = \mathsf{int}$, and $\rho(3.14) = \mathsf{double}$. The set of variables that are free in τ is $\mathit{fv}(\tau)$. $\top$ denotes the always satisfiable constraint and $\perp$ cannot be satisfiable by any ϕ, ψ, Θ.

Given two types τ and τ', constraint $\tau \equiv \tau'$ denotes, through a transitive relation, that these types are equivalent - note that when implicit conversions appear in a program, the involved types are considered to be equivalent (e.g., $\mathsf{int} \equiv \mathsf{char} \equiv \mathsf{float}$). Constraint $\tau \leq \tau'$ denotes a subtyping relation, which we use to model pointer semantics, as explained in Section 3.3. Constraint $\mathit{typeof}(x, \tau)$ expresses the fact that variable x, not necessarily declared, has type τ. We use constraint $\mathit{typedef}\ \tau\ \mathit{as}\ \tau'$ to indicate that type τ' is an alias of τ. Constraint $\mathit{has}(\tau, x : \tau')$ indicates that τ has a field of name x and type τ'. Symbol declarations and its types are registered using $\mathit{def}\ x : \tau\ \ \mathit{in}\ K$. We let $\psi \cup \{x \mapsto \tau\}$ denote the operation of updating finite map ψ with entry $x \mapsto \tau$. $K[\alpha \mapsto \tau]$ means variable substitution: we are replacing every occurrence of α in K with τ (implicit alpha-conversion avoids name capture). We let $\mathit{fields}(\tau)$ denote the set of fields in type definition τ. If τ is not a record type then $\mathit{fields}(\tau) = \emptyset$.

$$
\begin{array}{rcl}
\langle\langle\, \ell : \tau, \mathcal{M} \,\rangle\rangle_e & = & \rho(\ell) \equiv \tau \\
\langle\langle\, x : \tau, \mathcal{M} \,\rangle\rangle_e & = & \mathit{typeof}(x, \tau) \\
\langle\langle\, E\text{->}x : \tau, \mathcal{M} \,\rangle\rangle_e & = & \exists \alpha_1 \alpha_2 \alpha_3 . \langle\langle\, E : \alpha_1, \mathcal{M} \,\rangle\rangle_e \wedge \alpha_1 \equiv \alpha_2^* \wedge\ \mathit{has}(\alpha_2, x : \alpha_3) \wedge\ \alpha_3 \equiv \tau \\
\langle\langle\, *E : \tau, \mathcal{M} \,\rangle\rangle_e & = & \exists \alpha . \langle\langle\, E : \alpha, \mathcal{M} \,\rangle\rangle_e \wedge \alpha \equiv \tau^* \\
\langle\langle\, \&E : \tau, \mathcal{M} \,\rangle\rangle_e & = & \exists \alpha\, \alpha' . \langle\langle\, E : \alpha', \mathcal{M} \,\rangle\rangle_e \wedge \alpha \equiv \alpha'^* \wedge\ \alpha \equiv \tau \\
\langle\langle\, E = E' : \tau, \mathcal{M} \,\rangle\rangle_e & = & \exists \alpha_1 \alpha_2 . \langle\langle\, E : \alpha_1, \mathcal{M} \,\rangle\rangle_e \wedge \langle\langle\, E' : \alpha_2, \mathcal{M} \,\rangle\rangle_e \wedge \langle\langle\, E, \alpha_1, E', \alpha_2, \mathcal{M}, = \,\rangle\rangle_{kd} \\
\langle\langle\, E \oplus E' : \tau, \mathcal{M} \,\rangle\rangle_e & = & \exists \alpha_1 \alpha_2 . \langle\langle\, E : \alpha_1, \mathcal{M} \,\rangle\rangle_e \wedge \langle\langle\, E' : \alpha_2, \mathcal{M} \,\rangle\rangle_e \wedge \langle\langle\, E, \alpha_1, E', \alpha_2, \mathcal{M}, \oplus \,\rangle\rangle_{kd} \\
 & & \wedge\ \langle\langle\, \tau, \mathcal{M}(E \oplus E') \,\rangle\rangle_{sel}\ \wedge \mathit{typeof}(\oplus, \alpha_1 \to \alpha_2 \to \tau) \\
\langle\langle\, E; S, \tau, \mathcal{M} \,\rangle\rangle_s & = & \exists \alpha . \langle\langle\, E : \alpha, \mathcal{M} \,\rangle\rangle_e \wedge \langle\langle\, S, \tau, \mathcal{M} \,\rangle\rangle_s \\
\langle\langle\, \tau\ x; S, \tau, \mathcal{M} \,\rangle\rangle_s & = & \exists \alpha . \mathit{typedef}\ \tau\ \mathit{as}\ \alpha \wedge \mathit{def}\ x : \tau\ \mathit{in}\, \langle\langle\, S, \tau, \mathcal{M} \,\rangle\rangle_s \\
\langle\langle\, \tau\ f\ ((\tau^i\ x^i)^{i=1..n})\{S'\}; P, \mathcal{M} \,\rangle\rangle_d & = & \exists \alpha^{i=1..n} . (\mathit{typedef}\ \tau^{i=1..n}\ \mathit{as}\ \alpha^{i=1..n}) \wedge \mathit{def}\ f : \tau^{i=1..n} \to \tau\ \mathit{in} \\
 & & (\mathit{def}\ x^{i=1..n} : \tau^{i=1..n}\ \mathit{in}\, \langle\langle\, S', \tau, \mathcal{M} \,\rangle\rangle_s \wedge \langle\langle\, P, \mathcal{M} \,\rangle\rangle_d)
\end{array}
$$

Fig. 11. Constraint for expressions $\langle\langle\, E : \tau, \mathcal{M} \,\rangle\rangle_e$, statements $\langle\langle\, S, \tau, \mathcal{M} \,\rangle\rangle_s$ and declarations $\langle\langle\, D, \mathcal{M} \,\rangle\rangle_d$.

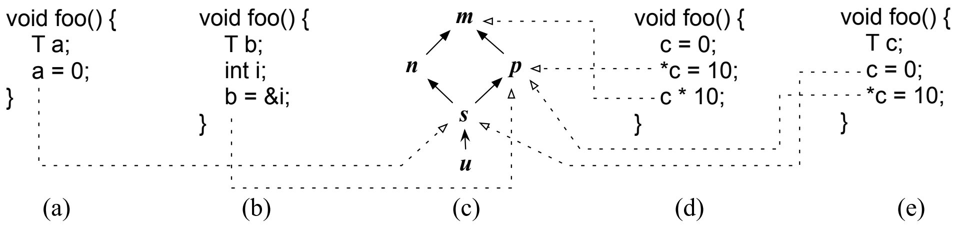
Syntax Directed Generation of Constraints. We generate constraints according to the rules in Figure 11. These rules define a visitor that traverses the AST produced in Section 3.1. For each node of this tree we generate different constraints. Three syntactic categories exist in the AST: expressions, statements and declarations. Rule $\langle\langle\, E : \tau, \mathcal{M} \,\rangle\rangle_e$ produces constraints for an expression E whose expected type is τ. Rule $\langle\langle\, S, \tau, \mathcal{M} \,\rangle\rangle_s$ produces constraints for a statement S within a function of return type τ. Rule $\langle\langle\, D, \mathcal{M} \,\rangle\rangle_d$ produces constraints for a declaration D.

All rules carry along a Table $\mathcal{M}$, which is populated by traversing expressions. But only two rules, which deserve special attention, consult this table[3]: binary expressions in general, $E \oplus E'$; and the assignment expression, specifically, $E = E'$ (from now on, we shall refer to this last one simply as assignment). These rules may involve types that are not necessarily unifiable. In a complete program, where semantic information is available, such types are distinguishable. However, due to missing declarations, we need an alternative way to ignore constraints that can lead to erroneous unification. Table $\mathcal{M}$, explained in the following section, helps us accomplish this task.

A Lattice for Constraint Generation. Apart from having an integral type in C, the value 0 is the *null pointer constant* [ISO-Standard 2011]{§6.3.2.3.3}. Hence, an assignment between a pointer and 0 is legal. However, we cannot unify these two types because the syntax of integers and pointers is not interchangeable. For instance, a must be a pointer in *a=10, and an arithmetic type in b * a. Therefore, our constraint generator must distinguish between pointer and integer variables. And yet, this distinction requires knowledge about type information, which might not be available in our case. To deal with this situation, we classify program expressions according to a lattice of pre-types $\mathcal{L}_M = \langle \{u, s, p, n, m\}, <, \vee, \bot = u, \top = m \rangle$, where *u=undefined*, *s=scalar*, *p=pointer*, *n=numeric*, and *m=malformed*. Every C type has a pre-type, but not all pre-types are C types. For instance, we might classify a particular expression as p even though the underlying type of the pointer is unknown. Non-scalar types are pre-typed as u. An undeclared variable pre-typed as n defaults to int. Fig. 12 (c) gives our lattice's partial order.

The classification of expressions, which happens prior to constraint generation, faces two complications. First, we may encounter names for which no declaration exists. Second, we may encounter names with an existing declaration whose type must be respected. For these reasons, our classification algorithm first attempts to categorize an expression by looking for declared types. Whenever it fails due to missing information, we employ the syntax-directed rules of Figure 13, which mimic restrictions imposed on actual C operators. These rules, for example, categorize obj as p if an expression obj->m is present in the program's text; and likewise categorize both x and y

[3] Our description is for μC. In actual C, matching of function arguments and formal parameters also need to be accounted.



```
void foo() {
  T a;
  a = 0;
}
```

```
void foo() {
  T b;
  int i;
  b = &i;
}
```

```
void foo() {
  c = 0;
  *c = 10;
  c * 10;
}
```

```
void foo() {
  T c;
  c = 0;
  *c = 10;
}
```

Fig. 12. The lattice $\mathcal{L}_{\mathcal{M}}$ of pre-types is in the center (c). (a,b,d,e) Mapping of expressions to points in $\mathcal{M}$. The order in which expressions are evaluated by the rules in Figure 13 is irrelevant.

as s if they appear in an x || y expression. This classification phase works as a “pre-inference”. It produces a map that associates every AST node that constitutes an expression with elements of $\mathcal{L}_{\mathcal{M}}$. We use $\mathcal{M}(node)$ to denote this mapping.

Once we have built $\mathcal{M}$, we consult it with two purposes. The first is to decide whether we keep or drop a constraint. When keeping a constraint, it is necessary to consider two cases. Binary expressions such as $E \oplus E'$ lead to *equivalence constraints* like $\tau_1 \equiv \tau_2$. Assignments such as $E = E'$ lead to *subtyping constraints* like $\tau_1 \leq \tau_2$ (the meaning of subtyping is further discussed in Section 3.3). Below we define a function $\langle\langle\, . \,\rangle\rangle_{kd}$, which consults $\mathcal{M}$:

$$\langle\langle\, E, \alpha, E', \alpha', \mathcal{M}, bin\_optr \,\rangle\rangle_{kd} = \begin{cases} \top, \text{ if } \mathcal{M}(E) \neq \mathcal{M}(E') \\ \alpha \equiv \alpha', \quad \text{if } bin\_optr \ is \ a \ binary \ expression(\oplus) \\ \alpha' \leq \alpha, \quad \text{if } bin\_optr \ is \ the \ assignment \ expression(=) \end{cases}$$

A constraint is dropped to avoid an incorrect unification arising from binary expressions. Four categories of binary operators, whose representatives are multiplication *, addition +, logical OR ||, and assignment =, are recognized. Figure 13 treats each category differently. Operators in the * category only apply to arithmetic types. In such cases, we can generate a type equivalence between the E and E'. Operators in the categories + and || can work with a mix of numeric values and pointers, and the generation of a type equivalence can trigger an undesired conversion. This risk of mismatch between arithmetic types and pointers exists with assignments as well, since 0 is an integer value, but also the *null pointer constant*.

Whenever one of the sides from an expression $E \oplus E'$ or $E = E'$ is ranked as p and the other as n, e.g., $\mathcal{M}(E) = p$ and $\mathcal{M}(E') = n$, we say that this expressions is *inconsistent*. A constraint associated with an inconsistent expression must be dropped, otherwise it will trigger an incompatible unification (called *over-unification* by Noonan *et al.* [Noonan et al. 2016]). Since this elimination occurs during generation stage, the effect to the solver is that the constraint never existed. Furthermore, n and p are ranked higher than s, so a constraint more restricting than the one being dropped must exist, unless the program is ill-typed (and a unification error will eventually be thrown). As Lemma 3.10 states, we do not lose information by dropping constraints. Example 3.9 illustrates an inconsistent expression, c = 0.

The second purpose for which we consult $\mathcal{M}$ is to select the right pre-type to annotate the result of a binary expression $E \oplus E'$; a action performed by the function $\langle\langle\, . \,\rangle\rangle_{sel}$. For assignments $E = E'$, this query is not necessary, since the resulting pre-type comes from the left-hand-side operand.

$$\langle\langle\, \tau, L \,\rangle\rangle_{sel} = \begin{cases} \tau \equiv \text{pointer}, \quad \text{if } L \ is \ p \\ \tau \equiv \text{numeric}, \quad \text{if } L \ is \ n \\ \top, \quad \text{otherwise (always satisfiable)} \end{cases}$$

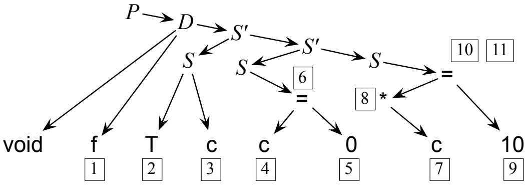
$$(x, L, \mathcal{M}) \rightarrow \mathcal{M} \cup \{x : L\} \qquad (\ell, L, \mathcal{M}) \rightarrow \mathcal{M} \cup \{\ell : n\} \qquad \frac{(E_1, L \vee p, \mathcal{M}) \rightarrow \mathcal{M}'}{(E_1\text{->}x, L, \mathcal{M}) \rightarrow \mathcal{M}' \cup \{E_1\text{->}x : u\}}$$

$$\frac{(E_1, L \vee s, \mathcal{M}) \rightarrow \mathcal{M}' \qquad (E_2, L \vee s, \mathcal{M}') \rightarrow \mathcal{M}"}{(E_1 || E_2, L, \mathcal{M}) \rightarrow \mathcal{M}" \cup \{E_1 || E_2 : \mathcal{M}"(E_1) \rightarrow \mathcal{M}"(E_2) \rightarrow n\}} \qquad \frac{(E_1, L \vee n, \mathcal{M}) \rightarrow \mathcal{M}' \qquad (E_2, L \vee n, \mathcal{M}') \rightarrow \mathcal{M}"}{(E_1 * E_2, L, \mathcal{M}) \rightarrow \mathcal{M}" \cup \{E_1 * E_2 : n \rightarrow n \rightarrow n\}}$$

$$\frac{(E, L \vee u, \mathcal{M}) \rightarrow \mathcal{M}'}{(\&E, L, \mathcal{M}) \rightarrow \mathcal{M}' \cup \{\&E : p\}} \qquad \frac{(E_1, L \vee s, \mathcal{M}) \rightarrow \mathcal{M}' \qquad (E_2, L \vee s, \mathcal{M}') \rightarrow \mathcal{M}"}{(E_1 + E_2, L, \mathcal{M}) \rightarrow \mathcal{M}" \cup \{E_1 + E_2 : \mathcal{M}"(E_1) \rightarrow \mathcal{M}"(E_2) \rightarrow (\mathcal{M}"(E_1) \vee \mathcal{M}"(E_2))\}}$$

$$\frac{(E, L \vee p, \mathcal{M}) \rightarrow \mathcal{M}'}{({}^*E, L, \mathcal{M}) \rightarrow \mathcal{M}' \cup \{{}^*E : u\}} \qquad \frac{(E_2, L \vee u, \mathcal{M}) \rightarrow \mathcal{M}' \qquad (E_1, L \vee \mathcal{M}'(E_2), \mathcal{M}') \rightarrow \mathcal{M}"}{(E_1 = E_2, L, \mathcal{M}) \rightarrow \mathcal{M}" \cup \{E_1 = E_2 : \mathcal{M}"(E_2) \rightarrow \mathcal{M}"(E_1) \rightarrow \mathcal{M}"(E_1)\}}$$

Fig. 13. Rules used to populate the table $\mathcal{M}$. These rules correspond to restrictions imposed by C on operands of such expressions.



$\boxed{1}$ *def* f : (void) *in*
$\boxed{2}$ *typedef* T *as* $\alpha_2 \wedge$
$\boxed{3}$ *def* c : α_2 *in*
$\boxed{4}$ *typeof*(c, α_3) $\wedge$
$\boxed{5}$ $\alpha_4 \equiv$ int $\wedge$
$\boxed{6}$ $\top \wedge$
$\boxed{7}$ *typeof*(c, α_7) $\wedge$
$\boxed{8}$ $\alpha_7 \equiv \alpha_5{}^* \wedge$
$\boxed{9}$ $\alpha_6 \equiv$ int$\wedge$
$\boxed{10}$ $\alpha_6 \leq \alpha_5 \wedge$
$\boxed{11}$ $\alpha_5 \equiv$ int

Fig. 14. Constraints produced for the program seen in Figure 2 (c). Numbers indicate which AST nodes have produced each constraint. Constraint 6 replaces an incorrect unification between $\alpha_3 \equiv \alpha_4$. Alpha creation is omitted to avoid cluttering.

Example 3.9. Figure 14 shows the constraints produced for the program in Figure 2 (c). One of them, e.g., $\top$ at node 6, is dropped to prevent unifying α_4 (int) and α_3 (int*).

Lemma 3.10. *Given an expression E in a μC program P, if* $\mathcal{M}(E)$ *is* p *(resp.* n*), then* $\langle\langle P, \mathcal{M} \rangle\rangle_d$ *generates constraints that bind E to a pointer (resp. a numeric) type.*

3.3 Challenge 3 – Handling Asymmetries in Type Equivalences

The C language lets programmers decorate a type with a *type qualifier* [ISO-Standard 2011]{§6.7.3}, such as const or volatile[4]. These qualifiers might cause assignments to be *unidirectional*, further complicating unification. For instance, the meaning of const is that a variable cannot be modified after initialization. Thus, in Figure 3 (a) it is possible to type T1 as int or as const int[5].

At first sight, the choice between int and const int is analogous in program (b). However, in this latter case, the only valid type for T2 is int. Typing T2 as const int yields an illegal program: the immutability promise made by b is broken in expression b = 10. This observation, together with the one made for program (a), leads us to the following conclusion: assignment of non-pointer types do not require that qualifiers are inferred. We call this the *qualifier-neutral* strategy.

Forgetting about const is a convenience also allowed in program (c) of Figure 3. However, w is now a pointer. This means that there exist two variables referring to the same memory location and our qualifier-neutral strategy cannot be used. Nevertheless, it is correct to discard const in

[4] There are differences between volatile and const. Program $\mathcal{P}$ = [void f() { int x; volatile int y; y = x; }] is valid, but had we used const, it would be invalid. Nevertheless, both qualifiers share typing rules and we focus the discussion on const.

[5] Functions need to be taken care of in a similar manner: T must be const int in program $\mathcal{P}$ = [void g(T* v); void f() { const int* p; g(p); }]. Arguments correspond to the right-hand-side of an assignment, while formal parameters to the left-hand-side (the return is handled analogously).

```
void f() {
   T1 x;
   const int* cp = x;
   int* ncp = x;
   T2 y = cp;
}
```

(a)

Goal: find types for the type variables in $\Psi = \{\texttt{x}\colon \alpha_2, \texttt{cp}\colon \alpha_3, \texttt{ncp}\colon \alpha_5, \texttt{y}\colon \alpha_7\}$.

$$\alpha_2 \equiv \alpha_4 \wedge \alpha_2 \equiv \alpha_6 \wedge \alpha_3 = \texttt{const int*} \wedge \alpha_5 = \texttt{int*} \wedge \alpha_3 \equiv \alpha_8 \wedge \alpha_4 \equiv \alpha_3 \wedge \alpha_6 \equiv \alpha_5 \wedge \alpha_8 \equiv \alpha_7$$

(b)

$$\mathrm{K} = \alpha_2 \equiv \alpha_4 \wedge \alpha_2 \equiv \alpha_6 \wedge \alpha_3 = \texttt{const int*} \wedge \alpha_5 = \texttt{int*} \wedge \alpha_3 \equiv \alpha_8$$
$$\mathrm{K}_{\leq} = \alpha_4 \leq \alpha_3 \wedge \alpha_6 \leq \alpha_5 \wedge \alpha_8 \leq \alpha_7$$

(c)

Fig. 15. (a) Partially available sources: we are missing definitions of T1 and T2. (b) Constraints produced by a classic type inference system. (c) The constraints that we produce.

this case due to a different reason: implicit conversions from non-qualified to qualified pointers are permitted, if the latter are more restricted than the former [ISO-Standard 2011]{§6.3.2.3-2}. Furthermore, expression *c = 10 in program (c) makes typing T3 as int* the only valid solution.

Finally, let us consider the program in Figure 3 (d), where, again, a common memory location is referenced by two different variables. However, inferring T4 without const yields an illegal program in this case. The incorrectness is due to a conversion from a qualified to a non-qualified pointer, which may only be done explicitly: for const, this would break the promise of immutability. Therefore, from this asymmetry in implicit conversions we can derive the following conclusion: assignment between pointers must account for type qualifiers. Specifically, qualification on the right-hand-side must be propagated to the left-hand-side. We call this the *qualifier-aware* strategy.

3.3.1 Subtyping and Unification. The behaviour of pointer-related implicit conversions mimics a subtyping relation. By interpreting τ^* as a subtype of const τ^*, we establish the partial order $\tau^* \leq$ const τ^*. This ordering means that a τ^* can be directly assigned to a const τ^*, but an assignment on the opposite direction requires an explicit cast. Even though it is counter-intuitive to think of non-constant pointers being a subset of constant pointers, this subtyping relation meets Liskov's substitution principle: a τ^* *can safely be used in a context where a* const τ^* is expected. The premise of this relation, sustained by the qualifier-neutral and qualifier-aware strategies, is the reason why, in Figure 11, we generate constraints for assignments in the form of type inequalities, instead of type equivalences. To make our discussion general, we now rephrase the qualifier-neutral and qualifier-aware strategies as *subtyping-neutral* and *subtyping-aware* strategies, respectively.

A subtyping relation originates from a type inequality constraint. Constraints, their syntax and semantics, were defined in section 3.2 as a first order logic model representing the type definitions of μC. To infer typing information that is missing from a partial program, we must solve those constraints. However, the presence of subtyping relations prevents us from accomplishing such a task via classical unification [Robinson 1965], which relies on the symmetry of type equivalences.

Example 3.11. Figure 15 (b) shows the constraints that would be produced for the program in Figure 15 (a) if we did not have inequality constraints. If we solved the constraints in Fig. 15 (b), then we would bind x to const int*, which is incorrect. This typing assignment is forbidden by the C standard. kcc [Ellison and Rosu 2012; Hathhorn et al. 2015; Inc. 2017], which adheres strictly to it, refuses the program. gcc and clang are more permissive: albeit with warnings, compilation succeeds. To deal with this problem, we separate constraints into two groups, K and $K_{\leq}$, as Figure 15 shows. These constraints shall be solved by the rules in Figure 16, which we explain in Section 3.4.

Unification and subtyping have been studied by Kaes [Kaes 1992] and Smith [Smith 1994]. In their systems, however, subtyping constraints are separately solved after types have been instantiated. Recently, Dolan and Mycroft [Dolan and Mycroft 2017] introduced *biunification*, to deal with subtyping from first principles. As a means to solve inequalities, Dolan and Mycroft define the notion of *positive* types, τ^+, and negative types, τ^-, which respectively correspond to output and input. In their system, constraints are of the form $\tau^+ \leq \tau^-$ and, instead of plain substitutions, biunification employs *bisubstitutions*. Bisubstitutions come with additional complexity: they apply independently on positive/negative types and result in an algebra of $\sqcap$ and $\sqcup$ lattice operators.

A Two-Phases Unification Approach. While Dolan and Mycroft's biunification solves μC's constraints, we chose to develop an alternative algorithm that deals with both type equivalence and inequality simultaneously, within a single solving method. Even though a type equivalence could be replaced by two type inequalities, type equivalences let us divise a simpler method to solve type inequalities, given our restricted form of subtyping. Our key idea to deal with subtyping without bisubstitutions is to perform unification twice, but each time with a different unification algorithm:

- $\mathcal{U}_c$ is essentially the classical unification [Martelli and Montanari 1982].
- $\mathcal{U}_s$ is a unification algorithm that incorporates the subtyping strategies we discussed.

Unification $\mathcal{U}_c$ is initially applied on type equivalences. On a purpose similar to that of Peyton Jones *et al.* [Peyton Jones et al. 2006], this allows us to benefit from concrete types' declarations which are available in the partial program. The substitutions produced are applied on the remaining constraints. Given that concrete types are now available, we used them to sort the inequality constraints according to a partial order. Sorting constraints produce a new list of constraints where positive types of a higher-order appear before negative types of a lower-order. As an example, when constraints are sorted, const $\tau^* \leq \alpha_x$ would be positioned in the list before constraint $\alpha_y \leq \tau^*$.

The fact that inequality constraints are ordered enables us to unify them, but now with unification $\mathcal{U}_s$. Even though the constraints being unified are that of inequalities, the subtyping relation will be respected, since the order in which constraints appear is the order in which type variables are bound, which, in turn, is determined by our partial order. Borrowing Dolan's notation, in $\mathcal{U}_s$, the subtyping-aware strategy becomes active for constraints involving τ^* as a *positive type*. Otherwise, when a τ^* is a *negative type*, subtyping-neutral strategy is the active one.

3.4 Challenge 4 – Solving Constraints

In section 3.2, we defined the syntax and the semantics of constraints as a first order logic model representing the type definitions of μC. To find typing information that is missing in a given μC program, we must solve those constraints – a task that we accomplish via first order unification [Robinson 1965]. Figure 16 shows a rewriting system that produces an equivalent constraint in a *solved form*. Constraints in solved forms are conjunctions of type equalities and field access relations. Notation $\overline{x}$ denotes sequences of elements from set $X = \{x_1, ..., x_n\}$. We abuse the notation to use set operations on sequences, e.g., $X \cap Y$. We remind the reader that $fv(\tau)$ is the set of variables that are free in τ. This notation lets us state Definition 3.12:

Definition 3.12 (Solved form). A solved form is a constraint $\exists\overline{\alpha}.(\overline{\beta \equiv \tau}) \wedge [\overline{has(\tau_k, x_k : \tau_k')}]$, where $fv(\overline{\tau}) \subseteq \overline{\alpha}$ and $\overline{\alpha} \cap \overline{\beta} = \emptyset$.

A solved form is a conjunction of type equivalences and field access constraints. It represents a solution to our type inference problem because: (i) type equivalences can be understood as type substitutions that unify, yielding most general types, and (ii) field access contain the information necessary to reconstruct the structure of missing records. Example 3.13 illustrates this definition.

Example 3.13. The solved form for the example seen in Figure 14 is: $\exists\alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7.\alpha_2 \equiv \alpha_7 \wedge \alpha_4 \equiv \mathsf{int} \wedge \mathsf{T} \wedge \alpha_2 \equiv \alpha_7 \wedge \alpha_7 \equiv {\alpha_5}^* \wedge \mathsf{int} \equiv \alpha_6 \wedge \alpha_5 \equiv \alpha_6 \wedge \alpha_5 \equiv \alpha_6$. Mapping back the solution of these constraints to the program in Figure 2, we conclude that T, the type of variable c, is int*.

To produce solved forms, the constraint solver of Figure 16 rewrites *solver configurations*. A configuration is a 4-tuple formed by finite mappings ψ and Θ, plus two constraints K and $K_{\leq}$. K is a

(a) *Structural Rules*

$\langle \psi;\ \Theta;\ (\exists\alpha.K_1) \wedge K_2;\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \exists\alpha.(K_1 \wedge K_2);\ K_\leq \rangle$ where $\alpha \notin fv(K_2)$
$\langle \psi;\ \Theta;\ K_1 \wedge (\exists\alpha.K_2);\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \exists\alpha.(K_1 \wedge K_2);\ K_\leq \rangle$ where $\alpha \notin fv(K_1)$
$\langle \psi;\ \Theta;\ K \wedge K';\ K_\leq \rangle \rightsquigarrow \langle \psi_2;\ \Theta_2;\ K_1 \wedge K_2;\ K_\leq \rangle$
where $\langle \psi;\ \Theta;\ K;\ K_\leq \rangle \rightsquigarrow \langle \psi_1;\ \Theta_1;\ K_1;\ K_\leq \rangle$ and $\langle \psi_1;\ \Theta_1;\ K';\ K_\leq \rangle \rightsquigarrow \langle \psi_2;\ \Theta_2;\ K_2;\ K_\leq \rangle$

(b) *Preprocessing Phase*

$\langle \psi;\ \Theta;\ K \wedge \mathit{typedef}\ \tau\ \mathit{as}\ \alpha \wedge K';\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta \cup \{\tau \mapsto \alpha\};\ K \wedge K';\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ K \wedge \mathit{def}\ \ x : \tau\ \wedge K';\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ (K \wedge K')[x \mapsto \tau];\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ K \wedge \mathit{typeof}(x,\tau) \wedge K';\ K_\leq \rangle \rightsquigarrow \langle \psi \cup \{x \mapsto \tau\};\ \Theta;\ K \wedge \psi(x) \equiv \tau \wedge K';\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ K \wedge \tau \equiv \tau' \wedge K';\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ K \wedge \Theta(\tau) \equiv \Theta(\tau') \wedge K';\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ K \wedge \tau \leq \tau' \wedge K';\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta; K \wedge K';\ K_\leq \wedge \Theta(\tau) \leq \Theta(\tau') \rangle$
$\forall i,j.\langle \psi;\ \Theta;\ K \wedge \mathit{has}(\tau, x : \tau_i) \wedge K' \wedge \mathit{has}(\tau, x : \tau_j) \wedge K'';\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau_i \equiv \tau_j \wedge K \wedge K' \wedge K'';\ K_\leq \rangle$

(c) *First Unification* ($\mathcal{U}_c$)

$\langle \psi;\ \Theta;\ \alpha \equiv \tau \wedge K;\ K_\leq \rangle \rightsquigarrow \langle \psi[\alpha \mapsto \tau];\ \Theta[\alpha \mapsto \tau];\ K[\alpha \mapsto \tau];\ K_\leq[\alpha \mapsto \tau] \rangle$
$\langle \psi;\ \Theta;\ \tau \equiv \alpha \wedge K;\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \alpha \equiv \tau \wedge K;\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ {\tau_1}^* \equiv {\tau_2}^* \wedge K;\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau_1 \equiv \tau_2 \wedge K;\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ \mathsf{const}\ \tau_1 \equiv \mathsf{const}\ \tau_2 \wedge K;\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau_1 \equiv \tau_2 \wedge K;\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ \tau \equiv \tau \wedge K;\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ K;\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' \wedge K;\ K_\leq \rangle \rightsquigarrow\ \langle \psi;\ \Theta;\ \tau_1 \equiv \tau_1' \wedge \tau_2 \equiv \tau_2' \wedge K;\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ \top \wedge K;\ K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ K;\ K_\leq \rangle$

(d) *Inequalities Ordering*

$\langle \psi;\ \Theta;\ K;\ K_\leq \wedge \mathsf{const}\ \mathbf{t}^* \leq \tau \wedge K_\leq' \rangle \rightsquigarrow \langle \psi;\ \Theta;\ K; \mathsf{const}\ \mathbf{t}^* \equiv \tau \wedge\ K_\leq \wedge K_\leq' \rangle$
$\langle \psi;\ \Theta;\ K;\ K_\leq \wedge \tau \leq \mathsf{const}\ \mathbf{t}^* \wedge K_\leq' \rangle \rightsquigarrow \langle \psi;\ \Theta;\ K;\ K_\leq \wedge K_\leq' \wedge\ \tau \equiv \mathsf{const}\ \mathbf{t}^* \rangle$
$\langle \psi;\ \Theta;\ K;\ K_\leq \wedge \tau \leq \tau' \wedge K_\leq' \rangle \rightsquigarrow \langle \psi;\ \Theta;\ K;\ K_\leq \wedge \tau \equiv \tau' \wedge K_\leq' \rangle$
$\langle \psi;\ \Theta;\ K;\ \tau \equiv \tau' \wedge K_\leq \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau \equiv \tau' \wedge K;\ K_\leq \rangle$
$\langle \psi;\ \Theta;\ K;\ \tau \equiv \tau' \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau \equiv \tau' \wedge K;\ \top \rangle$

(e) *Second Unification* ($\mathcal{U}_s$)

...
$\langle \psi;\ \Theta;\ \tau_1 \equiv \mathsf{const}\ \tau_2\ \ \wedge K;\ \top \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau_1 \equiv \tau_2\ \ \wedge K;\ \top \rangle$
$\langle \psi;\ \Theta;\ \mathsf{const}\ \tau_1 \equiv \tau_2\ \ \wedge K;\ \top \rangle \rightsquigarrow \langle \psi;\ \Theta;\ \tau_1 \equiv \tau_2 \wedge K;\ \top \rangle$

Fig. 16. Constraint solving split in five groups of rules. Rules for the second unification ($\mathcal{U}_s$) include all the rules in the first unification ($\mathcal{U}_c$), plus the two rules shown in part (e). In particular, the ellipsis, ..., indicate that the rules in part (e) pattern match after those in part (c).

conjunction of all the equivalence constraints, e.g., $\tau_1 \equiv \tau_2$. $K_\leq$ is a conjunction of all the inequality constraints, e.g., $\tau_1 \leq \tau_2$. The result of constraint solving is a *final configuration*, which is a solver configuration $\langle \psi;\ \Theta;\ K;\ K_\leq \rangle$ where K and $K_\leq$ are constraints in solved form or the unsatisfiable constraint $\perp$. Our solving process contains rules that are split into five groups:

- *Structural Rules* – Figure 16 (a): rules from this group are used at the beginning of the solving process. They introduce fresh type variables that are carried over across constraints.
- *Prepocessing Phase* – Figure 16 (b): these rules populate contexts with variables and types, and replace "*has*" constraints with equivalences relating fields of structs that should unify.
- *First Unification* – Figure 16 (c): these rules perform type inference proper, to unify type equivalences, through unification $\mathcal{U}_c$, which we have introduced in Section 3.3.1.
- *Inequalities Ordering* – Figure 16 (d): the constraints that constitute the conjunction $K_\leq$ need to be ordered before converted to standard equivalences, as we explained in Section 3.3.1. The rules from this group perform this sorting operation.
- *Second Unification* – Figure 16 (e): in this phase we conclude type inference. Because the ordering of constraints implies the partial order of our subtyping relation, type inequalities

```
void f() {
    T1 x;
    const int* cp = x;
    int* ncp = x;
    T2 y = cp;
}
```

Definitions after Preprocessing Phase
$\Psi = \{\text{x: }\alpha_2, \text{cp: }\alpha_3, \text{ncp: }\alpha_5, \text{y: }\alpha_7\}$
$K = \alpha_2 \equiv \alpha_4 \wedge \alpha_2 \equiv \alpha_6 \wedge \alpha_3 = \text{const int*} \wedge \alpha_5 = \text{int*} \wedge \alpha_3 \equiv \alpha_8$
$K_{\leq} = \alpha_4 \leq \alpha_3 \wedge \alpha_6 \leq \alpha_5 \wedge \alpha_8 \leq \alpha_7$

First Unification - Round 1
$\Psi = \{\text{x: }\alpha_2, \text{cp: }\alpha_3, \text{ncp: }\alpha_5, \text{y: }\alpha_7\}$
$K = \alpha_2 \equiv \alpha_4 \wedge \alpha_2 \equiv \alpha_6 \wedge \alpha_3 \equiv \text{const int*} \wedge \alpha_5 \equiv \text{int*} \wedge \alpha_3 \equiv \alpha_8$
$K_{\leq} = \alpha_4 \leq \alpha_3 \wedge \alpha_6 \leq \alpha_5 \wedge \alpha_8 \leq \alpha_7$

First Unification - Round 2
$\Psi = \{\underline{\mathbf{x: }\boldsymbol{\alpha_4}}, \text{cp: }\alpha_3, \text{ncp: }\alpha_5, \text{y: }\alpha_7\}$
$K = \alpha_4 \equiv \alpha_6 \wedge \alpha_3 \equiv \text{const int*} \wedge \alpha_5 \equiv \text{int*} \wedge \alpha_8 \equiv \alpha_3$
$K_{\leq} = \alpha_4 \leq \alpha_3 \wedge \alpha_6 \leq \alpha_5 \wedge \alpha_8 \leq \alpha_7$

First Unification - Round 3
$\Psi = \{\underline{\mathbf{x: }\boldsymbol{\alpha_6}}, \text{cp: }\alpha_3, \text{ncp: }\alpha_5, \text{y: }\alpha_7\}$
$K = \alpha_3 \equiv \text{const int*} \wedge \alpha_5 \equiv \text{int*} \wedge \alpha_8 \equiv \alpha_3$
$K_{\leq} = \underline{\boldsymbol{\alpha_6 \leq \alpha_3}} \wedge \alpha_6 \leq \alpha_5 \wedge \alpha_8 \leq \alpha_7$

First Unification - Round 4
$\Psi = \{\text{x: }\alpha_6, \underline{\mathbf{cp: const\ int*}}, \text{ncp: }\alpha_5, \text{y: }\alpha_7\}$
$K = \alpha_5 \equiv \text{int*} \wedge \underline{\boldsymbol{\alpha_8 \equiv} \mathbf{\ const\ int*}}$
$K_{\leq} = \alpha_6 \leq \text{const int*} \wedge \alpha_6 \leq \alpha_5 \wedge \alpha_8 \leq \alpha_7$

First Unification - Round 5
$\Psi = \{\text{x: }\alpha_6, \text{cp: const int*}, \underline{\mathbf{ncp: int*}}, \text{y: }\alpha_7\}$
$K = \alpha_8 \equiv \text{const int*}$
$K_{\leq} = \alpha_6 \leq \text{const int*} \wedge \underline{\boldsymbol{\alpha_6 \leq} \mathbf{\ int*}} \wedge \alpha_8 \leq \alpha_7$

First Unification - Round 6
$\Psi = \{\text{x: }\alpha_6, \text{cp: const int*}, \text{ncp: int*}, \text{y: }\alpha_7\}$
$K = \top$
$K_{\leq} = \alpha_6 \leq \text{const int*} \wedge \alpha_6 \leq \text{int*} \wedge \underline{\mathbf{const\ int*} \boldsymbol{\leq \alpha_7}}$

Inequalities Ordering
$\Psi = \{\text{x: }\alpha_6, \text{cp: const int*}, \text{ncp: int*}, \text{y: }\alpha_7\}$
$K = \top$
$K_{\leq} = \text{const int*} \leq \alpha_7 \wedge \alpha_6 \leq \text{int*} \wedge \alpha_6 \leq \text{const int*}$

Second Unification - Round 1
$\Psi = \{\text{x: }\alpha_6, \text{cp: const int*}, \text{ncp: int*}, \text{y: }\alpha_7\}$
$K = \text{const int*} \equiv \alpha_7 \wedge \alpha_6 \equiv \text{int*} \wedge \alpha_6 \equiv \text{const int*}$
$K_{\leq} = \top$

Second Unification - Round 2
$\Psi = \{\text{x: }\alpha_6, \text{cp: const int*}, \text{ncp: int*}, \underline{\mathbf{y: const\ int*}}\}$
$K = \alpha_6 \equiv \text{int*} \wedge \alpha_6 \equiv \text{const int*}$
$K_{\leq} = \top$

Second Unification - Round 3
$\Psi = \{\underline{\mathbf{x: int*}}, \text{cp: const int*}, \text{ncp: int*}, \text{y: const int*}\}$
$K = \text{int*} \equiv \text{const int*}$
$K_{\leq} = \top$

Second Unification - Round 4
$\Psi = \{\text{x: int*}, \text{cp: const int*}, \text{ncp: int*}, \text{y: const int*}\}$
$K = \top$
$K_{\leq} = \top$

Fig. 17. Solving constraints for the program displayed on the top-right corner. We omit Θ from the unification rounds, as it bears no effect on this example. We find that T1 $\equiv$ int$*$, and T2 $\equiv$ const int$*$.

can be unified just as if they represented type equivalences. In this process, we discard type qualifiers; hence, letting types bound to, say, const int*, unify with int*.

The constraint solver is called with an initial environment Θ_{init}, containing the definitions of builtin types (int, char, etc). We let $\rightsquigarrow^\star$ denote the reflexive, transitive closure of $\rightsquigarrow$. Theorem 3.14 states that our solver is confluent and strongly normalizing. A complete example of our solving process can be seen in Figure 17, which concludes Example 3.11.

Theorem 3.14 (Confluence and termination). *For any constraint K, there exists ψ ,Θ and K' such that $\langle\psi;\ \Theta_{init};\ K;\ \top\rangle \rightsquigarrow^\star \langle\psi;\ \Theta;\ K'; K_{\leq}\rangle$ and $\langle\psi;\ \Theta;\ K';\ K_{\leq}\rangle$ is a final configuration.*

Our solver preserves the semantics of constraints seen in Figure 10. To state such property, we define some notation – further used in μC's type system, in Section 3.4. A constraint K_1 entails a constraint K_2, written as $K_1 \Vdash K_2$, if and only if, for every ϕ, ψ and Θ, the assertion $\phi;\ \psi;\ \Theta \models K_1$ implies $\phi;\ \psi;\ \Theta \models K_2$. We write $K_1 \approx K_2$ (equivalence), if and only if, $K_1 \Vdash K_2$ and $K_2 \Vdash K_1$ holds.

Theorem 3.15 (Constraint solver soundness). *For all ψ, Θ, K, $K_{\leq}$, ψ', Θ', K', and $K'_{\leq}$, if $\langle\psi;\ \Theta;\ K, K_{\leq}\rangle \rightsquigarrow \langle\psi';\ \Theta';\ K', K'_{\leq}\rangle$ then $K \approx K'$.*

Dealing with insufficient information. Our solver is sound, but not complete. Some partial programs do not have enough information to instantiate all constraint variables to concrete μC types. We call such uninstantiated variables *orphans*. Definition 3.16 introduces this concept.

Definition 3.16 (Orphan variables). Let $S = \langle \psi ;\ \Theta ;\ K \rangle$ be a final solver configuration, such that $K = \exists\,\overline{\alpha}.[\overline{\alpha \equiv \tau}] \wedge [\overline{has(\tau, x : \tau)}]$ is satisfiable. Let $\phi = \overline{[\alpha \mapsto \tau]}$ be a finite mapping between type variables and types built from solved form type equivalences. The set of orphan variables of S, $fov(S)$, is defined as:

$$fov(S) = \left[\bigcup_{[x \mapsto \alpha] \in \psi} fv(\phi\ (\alpha)) \right] \cup \left[\bigcup_{[\tau' \mapsto \alpha] \in \Theta} fv(\phi\ (\alpha)) \right]$$

Example 3.17. The type variables associated with name T in Figures 1 (d) and T3 in Figure 4 (c) will be left a orphan and pointer to orphan, respectively, because none of these programs gives us any syntactic hint about the nature of these types.

As discussed in Challenge 4, uninstantiated variables cannot appear in reconstructed programs. Our approach to overcome this problem is inspired by Haskell 98. The Haskell 98 inference engine defaults uninstantiated kind variables, during kind inference, to ⋆, the Haskell kind for types [Faxén 2002; Peyton Jones et al. 2003]. Similarly, we have two defaulting rules, whose use is determined by two distinct cases: i) the entire declaration of an orphan variable is missing; or ii) only the declared type of an orphan variable is absent. An orphan variable that has no declaration may be instantiated with any type; we use int. Differently from that, if the declaration of an orphan variable is present, but the definition of its type is missing, then we instantiate it with a type that honours the *name* of the missing type. Example 3.18 illustrates both cases.

Example 3.18. Variable a is an orphan in program [void f(){a; }]. We instantiate it by creating a declaration of a with type int, due to defaulting rule (i). For an example of defaulting rule (ii), b is an orphan in program [void g(){struct T b; b; }]. To instantiate b, we define the type struct T as struct T {int dummy; }[6].

Orphan variables do not hinder our reconstruction process. According to Theorem 3.19, there is no flow of information, in Hunt-Sands sense [Hunt and Sands 2006], between orphan and non-orphan variables. Theorem 3.19 is equivalent to showing that: (i) orphans do not control conditional statements; (ii) orphans do not index memory accesses; and (iii) there is no assignment between orphan and non-orphan variable.

Theorem 3.19. *There is no flow of information between orphan and non-orphan variables.*

Variadic functions and generic selections. Variadic functions are inferred in the following manner. Constraints of function types are ordered by increasing number of arguments. During unification of those arguments, errors due to incompatible types or to inconsistent parameter-count are caught. Every function in ψ for which such errors are identified is made variadic - the ellipsis, ..., is placed at the index that triggered the error. Variadic functions such as those from the printf family can be registered into PsycheC so that the format-specifier string is used to determine the type of variadic arguments.

Relying on unification errors due to incompatible argument types helps us handle C11's generic selection [ISO-Standard 2011]{§6.5.1.1}. But since a variadic function must have at least one named argument [ISO-Standard 2011]{§6.7.6.3/5-9}, for an error occurring at the first index PsycheC

[6] A dummy field is created because the C standard requires that a struct has a non-empty field list [ISO-Standard 2011]{§6.7.2.1.8}.

$$\frac{}{K \mid \Gamma \vdash \ell : \rho(\ell)}\{Tl\} \qquad \frac{\begin{array}{c}K \mid \Gamma \vdash E : \tau' \\ K \Vdash \exists \alpha.\tau' \equiv \alpha{*} \wedge has(\alpha, x : \tau)\end{array}}{K \mid \Gamma \vdash E\text{->}x : \tau}\{Th\} \qquad \frac{K \mid \Gamma \vdash D \quad K \mid \Gamma \vdash P}{K \mid \Gamma \vdash D\,P}\{TDSeq\}$$

$$\frac{\begin{array}{c}K \mid \Gamma \vdash E : \tau_1 \qquad K \mid \Gamma \vdash E' : \tau_2 \\ \Gamma(\oplus) = \tau_1 \to \tau_2 \to \tau\end{array}}{K \mid \Gamma \vdash E \oplus E' : \tau}\{Tbop\} \qquad \frac{K \mid \Gamma \vdash E_1 : \tau \quad K \mid \Gamma \vdash E_2 : \tau' \quad K \Vdash \tau' \leq \tau}{K \mid \Gamma \vdash E_1 = E_2 : \tau}\{TAssign\}$$

$$\frac{\Gamma(x) = \tau \quad K \Vdash \exists \alpha.\alpha \equiv \tau}{K \mid \Gamma \vdash x : \tau}\{Tv\} \qquad \frac{K \mid \Gamma \vdash E : \tau' \quad K \Vdash \tau' \equiv \tau^{*}}{K \mid \Gamma \vdash {*}E : \tau}\{Tp\} \qquad \frac{K \mid \Gamma \vdash E : \tau' \quad K \Vdash \tau' \leq \tau}{K \mid \Gamma \vdash E : \tau}\{TSub\}$$

$$\frac{K \mid \Gamma \vdash_{\tau'} S \quad K \mid \Gamma \vdash_{\tau'} S'}{K \mid \Gamma \vdash_{\tau'} S\,S'}\{TSeq\} \qquad \frac{K \Vdash \exists \alpha.typedef\ \tau\ as\ \alpha \quad K \mid \Gamma, x : \tau \vdash_{\tau'} S'}{K \mid \Gamma \vdash_{\tau'}\ \tau\ x;\ S'}\{TLVar\}$$

$$\frac{\begin{array}{c}K \Vdash \exists \alpha.typedef\ \ \tau\ \ \alpha \qquad K \Vdash (\exists \alpha_i.typedef\ \tau_i\ \ as\ \ \alpha_i)^{i=0..n} \\ K \mid \Gamma, f : \tau_0, \ldots, \tau_n \to \tau,\ x_0 : \tau_0, \ldots, x_n : \tau_n\ \vdash_{\tau} S\end{array}}{K \mid \Gamma \vdash\ \tau\ f\ (\tau_i\ x_i)^{i=0..n}\{S\}}\{TFun\} \qquad \frac{K \mid \Gamma \vdash E : \tau}{K \mid \Gamma \vdash \&E : \tau^{*}}\{Td\}$$

Fig. 18. Typing relations for μC.

#defines a _Generic macro that forwards to artificial functions, one for each instantiated argument type. When _Generic appears in the program, either because the source was preprocessed or the keyword was directly used[7], we parse the call but constraints are not generated for arguments.

The Type System. Figure 18 shows type system rules for μC programs. Such rules are parameterized by a typing environment Γ, and a set of constraints K. Following standard practice, we let $\Gamma, x : \tau$ denote the inclusion of entry $x : \tau$ in typing context Γ and $\Gamma(x)$ represent the type associated with symbol x in Γ or a fresh type variable, if no such entry is found. Formally:

$$\Gamma, x : \tau = \{x : \tau\} \cup \{x' : \tau' \mid x' : \tau' \in \Gamma \wedge x' \neq x\}$$
$$\Gamma(x) = \begin{cases} \tau & \text{if } x : \tau \in \Gamma \\ \alpha \text{ fresh} & \text{otherwise} \end{cases}$$

Given a μC program p, we say that $K \mid \Gamma \vdash p$ is provable, if: (i) p is a well typed program in typing context Γ and (ii) constraint K is satisfiable. The typing judgement used in statements $(\vdash'_{\tau})$ is slightly different than the judgements used for expressions and declarations. We use $\vdash_{\tau}$ in statements because we need to record the return type τ of the function declaration that encloses the statement being typed. The τ in $K \mid \Gamma \vdash_{\tau} S$ indicates that statement S is well typed in K, Γ and it occurs in a function with return type τ.

A completeness theorem for constraint generation does not hold, since we cannot guarantee that a constraint generated for a partial program is entailed by some K that is used to construct a typing derivation for the original program. This happens because orphan variables may not be convertible to original program types. Nevertheless, we can provide a strong guarantee about the reconstructed programs. Theorem 3.20 states this guarantee. The theorem relates the constraint generation process of Section 3.2 and the type system rules of this section. According to the theorem, the generated constraints are sufficient to ensure a type system derivation according to rules in Figure 18.

[7] A generic selection is a primary expression [ISO-Standard 2011]{§6.5.1.6}. While normally used in macro definitions, the _Generic keyword is a compiler symbol, not a preprocessor one.

Theorem 3.20 (Soundness of constraint generation). *Let $\mathcal{P}_{par} = D\ P$ be a valid µC program and $K = \langle\langle\, \mathcal{P}_{par},\ \mathcal{M}\, \rangle\rangle_d$ its corresponding constraint system. If K is satisfiable, there exists Γ and K' s.t. $K' \mid \Gamma \vdash \mathcal{P}_{par}$ holds and K' is the solved form for K with all orphan variables instantiated.*

4 THE IMPLEMENTATION OF PSYCHEC

PsycheC consists of two components. Type inference is implemented in Haskell through a custom-made constraint solver, as described in Section 3.4. The AST visitor that generates constraints, mentioned in Section 3.2, is implemented in C++. The parser from which our AST is obtained is a modified and extended version of the parser from the Qt Creator IDE [Project 2017b].

The C Language. The first standardized version of C, published by ANSI, is known as C89 [ANSI-Standard 1989][8]. Since then, two major revisions of the language have been published by ISO. They are respectively known as C99 [ISO-Standard 1999] and C11 [ISO-Standard 2011]. Over time, C compilers introduced extensions to the language and non-ISO dialects emerged. Besides, the ISO standard does not specify a formal semantics for C. For that reason, alternative interpretations of the language's behaviour can been seen in different compilers. As a result, a survey conducted by Memarian *et al.* [Memarian et al. 2016] concludes that the expectations of C programmers, the assumptions made by static analysis tools, the behaviour of compilers, and the normative description from the ISO standard, diverge among themselves in several aspects.

Given the aforementioned subtleties, how accurately can we verify programs reconstructed by PsycheC? Our primary guidance is the ISO standard, but actual validation is done through compilers. A C11 compiler that attempts to rigorously conform to the standard is kcc [Ellison and Rosu 2012; Hathhorn et al. 2015; Inc. 2017]. It can diagnose issues that neither gcc, clang, nor icc, even in strict/pedantic mode, might detect. On the other hand, gcc and clang, which are free compilers, are more widely adopted in the industry. Therefore, while we test PsycheC with all those compilers, we ultimately strive for compliance with gcc and clang.

PsycheC covers almost all C99 language constructs. At the point of this writing, we lack designated initializers for arrays, compound literals, static array indices, and complex numbers. Although C11 brings significant features (e.g. memory model, multithreading, preprocessor-related and library additions), few of them relate to static semantics. So much of C11 is also understood. In its current form, PsycheC addresses the fundamental typing relations of C that are necessary for type inference. As shown in Section 5, we are capable of reconstructing the latest releases of many C libraries and incomplete source from popular open-source projects. Limitations and cases with special treatment are discussed in the following paragraphs.

Unexpanded Macros. The input of PsycheC is a *translation unit* [ISO-Standard 2011]{§5.1.1.1}, so the submitted incomplete source must have been preprocessed. One question arising from this scenario is: how to handle macros whose definitions are unavailable? The ideal case is when a macro, even in expanded form, conforms to C's grammar. Object- and function-like macros usually fit into this case, allowing successful parsing and a valid program reconstruction. For situations in which syntax errors appear due to unexpanded macros, PsycheC offers an extension-point that allows one to register predefined expansions - we observe that such macros frequently belong to a project's API, in which case this configuration can be done once and shared across developers. Nevertheless, we highlight that, among thousands of lines evaluated in Section 5, less than ten unique syntactically-invalid macros were found.

Example 4.1. In partial program $\mathcal{P}_1$ = [void g() { M_A(10); }], M_A is an unexpanded macro. Since $\mathcal{P}_1$ is accepted by C's grammar, PsycheC can parse it and infer M_A as a function. Partial

[8] C89 was later ratified by ISO and referred to as C90 [ISO-Standard 1990]. We disregard C95, since it is an amendment.

program $\mathcal{P}_2$ = [void f() { M_B(int, x) }], however, is invalid: we would be passing a type as an argument. But if a predefined expansion for `M_B(T, V)', such as `T V;', is registered in advance, PsycheC can reconstruct $\mathcal{P}_2$.

Declarations in C often appear surrounded by platform-specific decorators like a *calling conventions* specification (e.g. _cdecl), GNU's *__attribute__* specifiers, or Microsoft's *__declspec* import/export directives. Those decorators do not influence the typing relations of a partial program and are used by a compiler's backend for object-code generation. However, decorators do render an incomplete source invalid. PsycheC can handle them through empty builtin expansions configured on a platform-specific basis. This is the same mechanism employed by IDE's like Qt Creator to enable parsing (and, consequently, semantic-oriented features) in code editors.

Decaying: arrays x pointers, functions x function pointers. Our constraint's language does not distinguish array access from pointer expressions[9]. Due to the *decaying* rule [ISO-Standard 2011]{§6.3.2.1.3}, this inability is not a limitation. Contexts in which this differentiation matters, such as within the sizeof operator, affect dynamic semantics. Decaying from function to function pointers [ISO-Standard 2011]{§6.3.2.1.4} is handled by an *ad-hoc* unification rule that allows conversion between the two, combined with a late stage in our solving process that performs that decaying.

A Glimpse of Dynamic Semantics. Despite a varying degree of language-completeness coverage, there has been numerous studies on a formal dynamic semantics for C [Blazy and Leroy 2009; Ellison and Rosu 2012; Krebbers 2015; Krebbers and Wiedijk 2015; Papaspyrou 1998, 2001] and that of C's concurrency model [Batty et al. 2016; Nienhuis et al. 2016]. While the focus of our work is on producing well-typed programs, well-typedness does not eliminate the risk of *undefined behaviours* [Hathhorn et al. 2015]. PsycheC cannot guarantee, for a reconstructed program, that it behaves well at runtime. We do not offer this guarantee even when an incomplete source originates from a program free from undefined behaviours. Besides the fact that values may be absent, there exist other reasons that prevent us from establishing stronger guarantees about dynamic semantics: (i) inaccuracies on arithmetic types' inference can lead to signed integer overflows; (ii) fields of an inferred struct may be in different order from those in the original struct; (iii) missing array declarations are always inferred as pointers, possibly leading to unbound memory accesses; (iv) a function absent in the partial program has only its declaration inferred by PsycheC, not its definition - as mentioned at the end of Section 1, stub-generation tools [Cadar et al. 2008; Godefroid et al. 2005] can mitigate this problem. We notice that PsycheC's program reconstruction is cast-free: we do not introduce type casts in the program; hence, we do not change type relations already in place on the original code. In particular, we respect the contracts [Wadler and Findler 2009] between types and subtypes that we create to handle qualifiers. Finally, we emphasize that our results pertain to the static semantics of programs; hence, undefined behavior does not compromise our ability to find types for missing declarations.

5 EMPIRICAL EVALUATION

The key contribution of this paper is the technique to infer valid semantics to incomplete C sources. As we have seen in the previous sections, this endeavour implies no small amount of work, because C is not designed, from its beginning, to be amenable to type inference. Given this observation, why one would go over all the troubles to reconstruct missing declarations of C programs? The

[9] Array declarations are distinguishable, but in certain cases it is necessary to analyse further syntax: A bracket-less declaration like `T var;' could hide an array if a typedef like `typedef int T[2];' is missing.

answer to this question is another contribution of this paper. Thus, in the rest of this section, we describe practical uses of a type inference engine for incomplete C code.

The use cases that we shall discuss are not an exhaustive list of the possibilities that our ideas open up. PsycheC is a realistic, down-to-earth tool, with a community of users[10]. Since late 2016, PsycheC has been available through an online interface, where users upload their source, and get back a complete program. We know that this website has been used in different and, often, unexpected ways: as a code completion helper and as an assistant that reconstructs programs before they are forwarded to other tools.

5.1 Reconstructing Header Files

Goal: Show that we can reconstruct header files of real-world programs.
Motivation: When porting source code across platforms, it may happen that a software component depends on infrastructure that is not available on the target platform. For instance, during embedded software development, it can be the case that custom-hardware drivers cannot be compiled on traditional architectures, where we would like to run simulation or analyses. This was the original motivation for the development of PsycheC: to use Valgrind on software implemented for a particular embedded platform. PsycheC was used to aid porting those programs to Linux.
Benchmark: The 11 first programs (lexicographic order) from the latest version, 8.27, of the GNU Coreutils library - *change owner* appears twice because its implementation is split into two files. All headers, macro definitions, and top-level declarations are entirely removed from the source[11], the hardest setup for PsycheC's inference. Programs from GNU Coreutils are written in C99.
Discussion: Coreutils programs feature a rich set of C language constructs, an extensive variety of types, and broad coding style. Table 1 shows the result of our evaluation. Because we use an aggressive methodology to produce partial programs, the samples that we test have some of the ambiguous syntax seen in Figure 5. The parsing technique of Section 3.1 disambiguates some of them, as reported in column *Alg*. When further syntax is still not enough for us to resolve ambiguities, we resort to heuristics, following the approach of Knappen *et al* [Knapen et al. 1999]. Thus, x(y) is disambiguated as a function call; and x*y is disambiguated as a pointer declaration, for instance. Column *Heu* shows how often we resorted to heuristics. Our guesses turned out to be 100% correct for the Coreutils programs. This accuracy can be explained by the fact that those heuristics are based on common coding guidelines and constructs such as a multiplication with a discarded result is rare. Nevertheless, the algorithmic disambiguation presented in Section 3.1 is relevant to allow a formal end-to-end approach of our type inference. Table 1 also shows that constraint-solving time is proportional to the numbers of constraints, an expected result.

Both gcc and clang compile, without errors or warnings, the original programs. However, the original programs fail when compiled with kcc[12] because this compiler does not support the non-standard #include_next extension - trying to compile, with kcc, source preprocessed by other compiler does not work either, due to builtin expansions such as __builtin_va_list. On the other hand, kcc successfully compiles all the programs reconstructed by PsycheC.

Given that kcc is stricter than gcc and clang, it may come as surprise why the later diagnoses more warnings than the former. The reason for a large number of warnings by gcc and clang is because those two compiler detect if PsycheC redeclares a function or type that is part of C's standard library. As a matter of fact, the imprecision mentioned in the previous paragraph can render such redeclaration inconsistent with the one from the standard library. It is possible to run

[10] Earlier this year (2017), PsycheC appeared among GitHub's most trending projects.
[11] Invalid syntax due to unexpanded macros happened for the following macros: INT_BUFSIZE_BOUND, TYPE_SIGNED, GETOPT_HELP_OPTION _DECL, GETOPT_VERSION_OPTION_DECL, IF_LINT, SET_COMPONENT and _GL_UNUSED.
[12] Our kcc compilations have been performed through RV-Match, available at: https://runtimeverification.com/match/.

Table 1. Reconstruction of the GNU Coreutils programs. **LoC:** lines of code in the **Ori**ginal program and in the **Par**tial one; **Disamb:** syntactical ambiguities resolved **Alg**orithmically or **Heu**ristically; and **Sem**antic ambiguities resolved through our lattice; **Constr:** constraints of type **Eq**uivalences and of **Sub**typing, along with the **Time** (seconds) required to solve them; **gcc/clang/kcc:** number of **W**arnings and **E**rrors.

	LoC		Disamb			Constr			gcc		clang		kcc	
	Ori	Par	Alg	Heu	Sem	Eq	Sub	Time	W	E	W	E	W	E
base64.c	349	259	1	6	0	1,043	43	0.853	5	0	4	0	0	0
basename.c	189	150	3	6	2	503	32	0.279	6	0	4	0	0	0
cat.c	767	695	3	4	5	1,983	132	3.569	5	0	5	0	0	0
chcon.c	586	504	6	7	8	1,326	83	1.530	5	0	5	0	1	0
chgrp.c	318	244	1	5	2	734	43	0.484	11	0	12	0	0	0
chmod.c	569	468	3	7	10	1,435	123	1.757	6	0	9	0	0	0
chown.c	330	259	1	5	0	811	43	0.547	4	0	11	0	0	0
chown-core.c	554	507	9	1	2	1,932	143	3.233	3	0	3	0	0	0
chroot.c	429	366	8	4	15	1,349	98	1.675	14	0	15	0	2	0
cksum.c	318	249	2	4	1	768	40	0.564	2	0	2	0	1	0
comm.c	457	359	1	6	0	1,930	76	2.773	5	0	5	0	0	0

PsycheC in a *stdlib-compatible* mode so that it matches standard library names and uses the official declarations. But at this point, only part of the C's standard library has been implemented and this evaluation has been performed on the basis of "pure inference". Other sources of imprecisions of PsycheC that might trigger warnings are the following ones:

- Signed x unsigned mismatch: PsycheC cannot not always differentiate between undeclared signed and unsigned types; implicit conversions among them is permitted.
- Value is not an enumerator: upon switchs on enumerated types, gcc and clang might alert that an identifier in a case is not an enumerator. Even though the information that a name is an enumerator might not be inferable from syntax in this situation, PsycheC annotates the expression and its parts with a constexpr constraint, matching to C's *constant-expression* [ISO-Standard 2011]{§6.6} rule - a #define with an arbitrary value is generated.
- Unused expression result: due to unexpanded object-like macros in an expression-statement.

5.2 Enabling Static Analyses

Goal: Give static analyses tools the means to handle programs partially available.
Motivation: Prominent static analysis tools, such as SonarQube, OClint, Splint [Evans 1996; Larochelle et al. 2001], PVSStudio, clangStaticAnalyser, Checkmarx, Coverity, Klocwork and Frama-C [Cuoq et al. 2012] require full source files. They usually integrate with the build system. Analyzing cross-platform and embedded software can be arduous in this scenario. In fact, few of the aforementioned tools offer versions for Windows, Linux, and OSX. The industry tries to mitigate this problem with component-packages and plugin-based services. However, it is difficult to provide such support for every conceivable system. As consequence of these shortcomings, many static analysis tools cannot handle partial programs: they skip source sections or break down, when absent declarations are encountered. Either way, a diagnostic cannot be produced.
Benchmark: PVS-Studio[13], a tool that detects bugs in C, C++ and C# programs, and that works for Windows and Linux. The PVS-Studio website contains a vast suite of code snippets from popular open-source projects. But in order to analyze them, PVS-Studio needs the entire program. We have reconstructed many of those partial programs and submitted them to static analysis.
Discussion: Figure 19 shows the types that PsycheC reconstructs to three snippets taken, *as-is*, from PVS-Studio's show-case. Each of these examples illustrates a particular issue that PVS-Studio

[13] Frontpage at https://www.viva64.com/en/pvs-studio/; show-case examples at https://www.viva64.com/en/inspections/

a)

```
int _PyState_AddModule(PyObject* module, struct PyModuleDef* def) {
   PyInterpreterState *state;
   if (def->m_slots) {
      PyErr_SetString(PyExc_SystemError, "PyState...");
      return -1;
   }
   state = GET_INTERP_STATE();
   if (!def) return -1;
   //...
}
```

```
struct PyModuleDef { int m_slots; } ;
typedef int PyObject ;
typedef int  PyInterpreterState ;
int * GET_INTERP_STATE () ;
int PyErr_SetString (int ,char*) ;
int PyExc_SystemError;
```

b)

```
bit32 siHDAMode_V() {
  if(saRoot->memoryAllocated.agMemory[i].totalLength > biggest) {
    if(biggest < saRoot->memoryAllocated.agMemory[i].totalLength) {
       save = i;
       biggest = saRoot->memoryAllocated.agMemory[i].totalLength;
    }
  }
}
```

```
typedef struct TYPE_6__ TYPE_3__;
typedef struct TYPE_5__ TYPE_2__;
typedef struct TYPE_4__ TYPE_1__;
typedef int bit32;
struct TYPE_5__ { TYPE_1__* agMemory; };
struct TYPE_6__ { TYPE_2__ memoryAllocated; };
struct TYPE_4__ { int totalLength; };
int biggest, i, save;
TYPE_3__* saRoot;
```

c)

```
type_p find_structure (const char *name, enum typekind kind) {
   structures = s;  // assignment
   s->kind = kind;
   s->u.s.tag = name;
   structures = s;  // re-assignment
   return s;
}
```

```
/* Forward declarations omitted due to space */
typedef TYPE_3__* type_p;
typedef enum typekind {
      ____Placeholder_typekind } typekind;
struct TYPE_7__ {char const* tag; };
struct TYPE_8__ {TYPE_1__ s; };
struct TYPE_9__ {int kind; TYPE_2__ u; };
TYPE_3__* s, * structures;
```

Fig. 19. On the left, snippets from open-source projects. On the right, types inferred by PsycheC which preserve the same issues diagnosed for the complete program by PVS-Studio. (a) CPython: An if condition checks validity of the pointer def. However, this pointer is dereferenced in a previous if through access to field def->m_slots, potentially causing a segmentation fault. (b) FreeBSD: Nested if conditions with semantically equal expressions, only that the operator is inverted and the operands are at opposite sides. The conditions are redundant. (c) gcc: Successive assignments of the structures variable. One of them is meaningless.

finds automatically. *The program we reconstruct is diagnosed with the same issues as the original programs.* In spite of that, a program reconstructed by PsycheC does not, necessarily, contain all issues that would have been diagnosed for the original program. For instance, PsycheC might not differentiate between a signed and an unsigned arithmetic type (due to an implicit conversion), or, in the absence of a value that indexes an array, a buffer overrun may not be detectable. But in many situations, the cause of a diagnostic lies on the structure of a program. In cases similar to the snippets from Figure 19, PsycheC produces declarations that lead to the same diagnostics.

5.3 Improving Static Analyses

Goal: Eliminate false-positives from static analyses tools.
Motivation: Some static analyses tools employ a variation of fuzzy parsing [Koppler 1997] to deal with partial programs. An appealing advantage of this approach is that it requires "zero setup". Zero setup offers opportunities for broader use-cases: a developer can analyze source regions within a code editor, or individual functions extracted from a VCS (*Version Control System*), or code snippets submitted to a bug-tracker. However, without the aid of a type inference such as the one we propose, the zero setup scenario becomes less likely to be explored, since precision of the analysis degrades and the number of false-positives increases.

```
static int check_header_line(struct apply_state *state, struct patch *patch) {
  int extensions = (patch->is_delete == 1) + (patch->is_new == 1) +
                   (patch->is_rename == 1) + (patch->is_copy == 1);
  if (extensions > 1)
     return error(_("inconsistent header lines %d and %d"),
                  patch->extension_linenr, state->linenr);
  if (extensions && !patch->extension_linenr)
     patch->extension_linenr = state->linenr;
  return 0;
}
```

```
struct patch {
        int is_delete;
        int is_new;
        int is_rename;
        int is_copy;
        char* extension_linenr; };
struct apply_state {
        char* linenr; };
```

Fig. 20. On the left, a function introduced as a patch to the git project. On the right, types inferred by PsycheC which allowed PathCrawler to generate test-input data for a conclusive verdict of correctness.

Benchmark: Cppcheck[14], a static analyzer for C and C++, which detects errors such as out-of-bounds memory accesses, memory leaks and null pointer dereferences, for instance.
Discussion: Consider the hand-picked program $\mathcal{P}$ = [void f() { x b = 1; a ∗ b; ++b; }]. When processing this program, Cppcheck produces at, ++b, a false-positive diagnostic due to the use of "uninitialised variable b". This error happens because it cannot distinguish that a ∗ b must be a multiplication, not a declaration. CppCheck could benefit from a tool like PsycheC by reconstructing this program prior to the analysis, in which case the false-positive could be eliminated.

5.4 Supporting Software Testing

Goal: Enable, from isolated functions, the generation of stubs to test programs.
Motivation: A number of stub-generators, capable of fabricating meaningful test-input data, have been proposed to support software testing. Examples include KLEE [Cadar et al. 2008], PathCrawler [Williams et al. 2005], DART [Godefroid et al. 2005], and PEX [Tillmann and De Halleux 2008]. However, these tools do not address a practical aspect of testing complex systems: the ability to decouple, at the source level, functions of interest from their dependencies. This possibility makes testing more convenient and accessible. Thus, the aforementioned tools still require a complete program, either to be statically-analyzed or symbolically-executed.
Benchmark: PathCrawler, a tool that automatically generates test inputs for functions written in C. We use the version of PathCrawler available through an online interface[15]
Discussion: The function displayed in Figure 20 has been submitted as a patch[16] to the *git project*. The purpose of check_header_line is to enforce that no two operations such as adding, removing, copying, or renaming a file can happen simultaneously when a user issues command git commit. It does not scale to run tools such as PathCrawler on the entire program for every single commit. But to quickly provide preliminary feedback to a developer, we wish to generate test-input data for check_header_line and verify whether its implementation is correct.

PsycheC lets us solve this problem. From the isolated function, we infer types struct patch and struct apply_state; hence, enabling the compilation of function check_header_line. We submitted the reconstructed program to PathCrawler, along with a context and an oracle definition. PathCrawler could generate all test-input data we expected (sixteen cases, corresponding to the combinations of flags is_delete, is_rename, is_new, and is_copy) and of emitting a successful verdict for the function implementation. PsycheC can be used to help testing patches whenever a function appears in its entirety, the reason for which we picked this example.

[14] Available at http://cppcheck.sourceforge.net/ on July 2017
[15] Available at http://pathcrawler-online.com:8080/ on July 2017.
[16] Available at https://github.com/git/git/commit/d70e9c5c8c865626b6e69c2bf9fd0e368543617b

5.5 Extracting Data-Structures

Goal: Extract complete definitions of data structures from real-world software libraries.
Motivation: Many libraries provide, today, essential data structures, such as lists, binary trees, and hash tables. But relying on external libraries can be undesirable, due to dependency management or due to the sheer size and complexity of the library. The issue becomes more exacerbated if only a tiny portion of source code is to be reused. Under these circumstances, copying-and-pasting can be a workaround. However, such a manual process is error-prone and requires significant effort to navigate through the sources in order to hand-pick only the necessary parts of an implementation.
Benchmark: Functions that manipulate Abstract Data Types from the following industry-quality open-source libraries: GNOME's *GLib* [Project 2017a], the GNU Portability Library *Gnulib* [Foundation 2017] and the Generic Data Structure Library (*GDSL*) [Authors 2017]. In addition, we considered functions from Sedgewick's book [Sedgewick 2002][17]. We select as targets only functions that comprise the API of a basic "insert" operation (e.g. inserting an item into a list, inserting an item into a tree, etc) and consider the availability of data-structures in each library.
Discussion: We establish the following reconstruction criteria: by starting with a single function, we continuously add more of them until we are able to reconstruct at least 60% of the original data structure. It is in the nature of our technique that, the more we see of an incomplete source, the more accurate becomes the inference. For this experiment, a few macros, which were expanded, appear in the *slice*: function-like macros to conveniently access fields of complex *struct*s. In this experiment, PsycheC infers quite complex types. All programs we reconstruct compile successfully on gcc and clang. Kcc compiles all, but one of them: Gnulib's reconstructed hash table. When compiling it, kcc terminates without issuing any message.

Table 2 gives us an idea of how much of a data structure PsycheC can reconstruct from just a few functions – the exact number of them are indicated in column *Slice size*. The *Exact* matches correspond to fields inferred as the same type as in the original library's declaration. Others have either been *Conv*erted (e.g. between int and long) or identified as *Scalar*, because the available syntax was not enough to differentiate an integer from a pointer (e.g. a variable initialized with 0). Fields in the original library's implementation that do not appear in the slice are marked as *Unavail*able. An interesting observation about Table 2 is that inference from all implementations result in at a least one *Orphan* or *Partial* type, a type which we can only partially infer. For instance, a field inferred as a function pointer but whose return type is unknown. Another example is a pointer, but with unknown underlying type. Orphans and partial types typically correspond to the item being inserted, which, for extensibility purposes, is an opaque pointer (a pointer to an unspecified type) or comes from a user-supplied function-pointer or typedef.

The implementation style of ADTs also vary across libraries. While some of them use a single struct with all the fields, others split the ADT representation across two structs: typically, one for the node representation and another with fields that support the provided operations. When comparing the complexity of Sedgewick's textbook implementations against the industrial ones, the greatest difference appears with hash tables: for an open-addressing scheme, Sedgewick stores the table in a simple array; the professional libraries employ more advanced techniques. Hash tables are also the most challenging ADTs to reconstruct, among the ones we evaluated. They typically involve a larger number of fields, more opaque data, and many user-supplied function to calculate keys, define item equality, allocate notes, etc. The GDSL implementation, in particular, uses shorts for some internal types, which we infer as ints, reducing the number of exact matches. In regards

[17] The functions we shall use are available online in the following websites: https://www.cs.princeton.edu/~rs/Algs3.c1-4/code.txt and https://www.cs.princeton.edu/~rs/Algs3.c5/code.txt

Table 2. Field reconstruction of ADTs from different implementations. **Fields**: fields **Used** and **Unav**ailable in the slice taken; **Inferred**: types inferred **E**xactly; implicitly **C**onverted; as **S**calars, when syntax does not differentiating between a pointer or an integer; only **P**artially, such as a pointer which we do not know the underlying type; and **O**rphans; **Slice Size**: number of API functions that compose the slice. The last columns show the result of compiling our reconstructed programs with gcc, clang, and kcc.

		Fields		Inferrence					Slice	Compiler		
	ADT	Used	Unav	E	C	S	P	O	size	gcc	clang	kcc
GLib	Doub. Link. List	3	0	2	0	0	0	1	1	ok	ok	ok
	Queue	5	0	4	0	0	0	1	1	ok	ok	ok
	AVL Tree	12	2	3	3	1	2	3	3	ok	ok	ok
	Hash Table (open addr.)	8	5	2	0	2	4	0	1	ok	ok	ok
GDSL	Doub. Link. List	6	0	4	0	1	1	0	2	ok	ok	ok
	Queue	6	3	3	0	1	1	1	3	ok	ok	ok
	BST Tree	9	1	4	0	1	3	1	5	ok	ok	ok
	RB Tree	7	4	4	0	1	0	2	4	ok	ok	ok
	Hash Table (chain.)	9	1	0	2	0	6	1	1	ok	ok	ok
Gnulib	Link. List	3	3	1	0	1	1	0	1	ok	ok	ok
	AVL Tree	7	1	4	2	0	1	0	2	ok	ok	ok
	RB Tree	7	1	1	0	1	1	0	2	ok	ok	ok
	Hash Table	10	5	2	5	2	2	1	1	ok	ok	Unav
Sedgewick	Sing. Link.List	2	0	1	0	0	0	1	2	ok	ok	ok
	Priority Queue	4	1	3	0	0	0	1	1	ok	ok	ok
	Splay Tree	4	0	3	0	0	0	1	4	ok	ok	ok
	RB Tree	4	0	3	0	0	0	1	4	ok	ok	ok
	Hash Table (chain.)	3	0	2	0	0	0	1	2	ok	ok	ok
	Hash Table (open addr.)	1	0	0	0	0	1	0	1	ok	ok	ok
	Graph (adj. matrix)	3	0	2	0	1	0	0	2	ok	ok	ok
	Graph (adj. list)	5	0	4	0	1	0	0	3	ok	ok	ok

to the graph evaluation, we considered as an "insert" operation both the API to insert vertices and to insert edges into the graph.

6 RELATED WORK

The work that we have presented in this paper is built on contributions from a long string of research in programming languages, which we highlight in this section. However, we emphasize that, to the best of our knowledge, the compilation of incomplete C code, with all the challenges that this task entails, was thus far an unsolved problem.

Parsing of incomplete sources. There exists a body of work about parsing C/C++ in face of missing program parts [Bischofberger 1993; Knapen et al. 1999; Koppler 1997; Padioleau 2009]. The work of Knapen *et al.* [Knapen et al. 1999] relates closely to ours. They generate multiple AST nodes, until they have enough syntactic information to decide which one is valid. However, contrary to our work, neither Bischofberger *et al.* [Bischofberger 1993], Koppler *et al.* [Koppler 1997], Knapen *et al.* [Knapen et al. 1999], nor Padioleau [Padioleau 2009] reconstruct C programs. They parse programs to support static analyses, but the production of compilable code is not among their objectives. Hence, they do not deal with type and array size inference. Furthermore, none of these previous work formalize their notion of ambiguity resolution; thus, it is difficult to state precisely how they differ from our approach at the operational level. Therefore, the formalization of section 3.1 is a contribution of this work.

Type inference. Type inference is a staple feature in statically typed functional programming languages. Our approach is implemented after the HM(X) algorithm by Pottier and Remy [Pottier and Rémy 2005]: with separate stages for constraint generation and type inference properly. The main difference between the type inference that we do, and the one performed in languages such as

ML, is the fact that we do not have the declaration of algebraic types to guide the inference process. On the contrary, we are trying to reconstruct them. Noonan *et al.* [Noonan et al. 2016] provide a type inference algorithm for machine code that relates to our work. Their tool, *Retypd*, works in a two-phase approach: 1) a sound inference algorithm constructed over a type-system richer and more powerful than the actual C type-system; 2) a heuristic-based translation mechanism that converts types from such type-system to C source. Although similar to the techniques that we introduce in Sections 3.2 and 3.4, Noonan *et al.*'s method does not solve the problem that we address: *Retypd* starts with a complete binary, infers types to it, and reports possible C types that could have existed in the original source. We start with an incomplete C source, and recover the missing code, including type declarations. As explained in Section 3.3, we use a subtyping relation to model C pointers. Handling subtyping through unification is challenging due to the asymmetry of type equivalences. To this end, we were inspired by the biunification algorithm from Dolan *et al.* [Dolan and Mycroft 2017]. In particular, with the notion of input and output types and how to represent them in terms of inequalities. However, our technique is based on plain unification and does not employ bisubstitutions, yielding a simpler algorithm.

Program synthesis. There exists an enormous body of literature related to the synthesis of programs [Manna and Waldinger 1980]. The synthesis community relies on examples or high-level specifications to generate algorithms. We also want to generate code; however, we do not want to generate behavior. Thus, we reconstruct missing declarations, but not missing function bodies.

7 FINAL THOUGHTS

This paper provides a technique to support the compilation of incomplete C code. We showed how to recover missing syntactic and semantic information from sources partially available. In this process, we solved problems related to ambiguities that surface once declarations are missing and we overcame difficulties with C's typing rules, by extending a well-known type inference approach. Our technique lets us infer the types of variables and reconstruct a partial program into a new, complete one, that passes C's type-checker. The tool developed to sustain our ideas, PsycheC, is the first type inference engine for C. We have used PsycheC to illustrate the many possibilities that our theory opens up for researchers and practitioners.

Future Work. We believe that a general methodology to infer types for partially available C code opens up several avenues along which further research can be pursued. As an example, currently we do not associate size information with array types, they are made plain pointers like int*. Yet, current state-of-the-art symbolic range analysis, à la Nazaré *et al.* [Nazaré et al. 2014] should let us associate conservative size expressions with such type; hence, giving us int[42] or int[N+M], for instance. As another example of future work, it might be possible to establish guarantees about the dynamic semantics of the partial program to be reconstructed. Consider the case of signed integer overflows, for instance. Can we use standard static analyses [Rodrigues et al. 2013] to determine the exact type of scalars that are used in signed arithmetic operations amenable to overflows? Finally, while developing code with the support of PsycheC, programmers have at their disposable effectively a new programming language, which reuses C's syntax, but that supports type inference. The impact of this "new language" onto the productivity of C programmers is yet to be assessed. These, among other stories, we hope to discuss in the future.

Software. PsycheC's online interface is publicly available at http://cuda.dcc.ufmg.br/psyche-c. This webpage receives syntactically valid partial C code, and gives back the declarations that make that code compilable. The implementation of PsycheC is publicly available at https://github.com/ltcmelo/psychec under an open-source license.

ACKNOWLEDGEMENTS

We thank Alisa Maas, Jens Palsberg and our co-workers in the Compilers Lab for discussing an earlier version of this manuscript with us. We thank the POPL referees, and Xavier Leroy – who shepherded this paper – for the many suggestions and comments they have sent us. This research has been made possible through the support of CNPq, CAPES and FAPEMIG.

REFERENCES

ANSI-Standard. 1989. ANSI X3.159-1989 - The C Programming Language.

The GDSL Authors. 2017. The Generic Data Structures Library. http://home.gna.org/gdsl/.

Mark Batty, Alastair F Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *POPL*, Vol. 51. ACM, 634–648.

Walter R. Bischofberger. 1993. Sniff: a pragmatic approach to a C++ programming environment (abstract). *OOPS Messenger* 4, 2 (1993), 229.

Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*. USENIX, 209–224.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for javascript. In *PLDI*. ACM, 50–62.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International Conference on Software Engineering and Formal Methods*. Springer, 233–247.

Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. In *OOPSLA*. ACM, 313–328.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *POPL*. ACM, 1–13.

Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *POPL*, Vol. 47. ACM, 533–544.

David Evans. 1996. Static Detection of Dynamic Memory Errors. In *PLDI*, Vol. 31. ACM, 44–53.

Karl-Filip Faxén. 2002. A Static Semantics for Haskell. *J. Funct. Program.* 12, 5 (2002), 295–357.

The Free Software Foundation. 2017. Gnulib - The GNU Portability Library. https://www.gnu.org/software/gnulib/.

Patrice Godefroid. 2014. Micro Execution. In *ICSE*. ACM, 539–549.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. ACM, 213–223.

Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the Undefinedness of C. In *PLDI*. ACM, 336–345.

Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *POPL*. ACM, 79–90.

Runtime Verification Inc. 2017. RV-Match. https://runtimeverification.com/match/.

ISO-Standard. 1990. ISO/IEC 9899:1990 - The C Programming Language.

ISO-Standard. 1999. ISO/IEC 9899:1999 - The C Programming Language.

ISO-Standard. 2011. ISO/IEC 9899:2011 - The C Programming Language.

Stefan Kaes. 1992. Type inference in the presence of overloading, subtyping and recursive types. In *ACM SIGPLAN Lisp Pointers*. ACM, 193–204.

Gregory Knapen, Bruno Laguë, Michel Dagenais, and Ettore Merlo. 1999. Parsing C++ Despite Missing Declarations. In *IWPC*. IEEE, 114–125.

Rainer Koppler. 1997. A Systematic Approach to Fuzzy Parsing. *Softw. Pract. Exper.* 27, 6 (1997), 637–649.

Robbert Krebbers. 2015. *The C Standard Formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen.

Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*. ACM, 15–27.

David Larochelle, David Evans, et al. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities.. In *USENIX Security Symposium*, Vol. 32. Washington DC.

Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *TOPLAS* 2, 1 (1980), 90–121.

Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 2 (1982), 258–282.

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1–15.

Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2014. Validation of memory accesses through symbolic analyses. In *OOPSLA*. ACM, 791–809.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2005. *Principles of program analysis.* Springer.

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. In *OOPSLA*. 111–128.

Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic type inference for machine code. In *PLDI*. ACM, 27–41.

Yoann Padioleau. 2009. Parsing C/C++ code without pre-processing. In *CC*. Springer, 109–125.

Nikolaos S Papaspyrou. 1998. *A Formal Semantics for the C Programming Language.* Ph.D. Dissertation. National Technical University of Athens. Athens (Greece).

Nikolaos S Papaspyrou. 2001. Denotational Semantics of ANSI C. *Computer Standards & Interfaces* 23, 3 (2001), 169–185.

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed Completion of Partial Expressions. In *PLDI*. ACM, 275–286.

Simon Peyton Jones et al. 2003. The Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming* 13, 1 (Jan 2003), 0–255.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP*, Vol. 41. ACM, 50–61.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489.

The GNOME Project. 2017a. The GNOME Library - GLib. https://developer.gnome.org/glib.

The Qt Project. 2017b. The Qt Creator IDE. https://www.qt.io/ide/.

Didier Rémy. 2013. Type Systems for Programming Languages.

J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (1965), 23–41.

Raphael Ernani Rodrigues, Fernando Magno Quintao Pereira, and Victor Hugo Sperle Campos. 2013. A Fast and Low-overhead Technique to Secure Programs Against Integer Overflows. In *CGO*. IEEE, Washington, DC, USA, 1–11.

Robert Sedgewick. 2002. *Algorithms in C (3rd Edition).* Addison Wesley.

Geoffrey S Smith. 1994. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming* 23, 2-3 (1994), 197–226.

Leon Sterling. 1994. *The Art of Prolog* (2nd ed.). MIT Press.

Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *TAP*. Springer, 134–153.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can'T Be Blamed. In *ESOP*. Springer, 1–16.

Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. 2005. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *EDCC*. Springer, 281–292.