TYPE INFERENCE FOR C: APPLICATIONS TO THE ANALYSIS OF INCOMPLETE PROGRAMS

LEANDRO T. C. MELO

TYPE INFERENCE FOR C: APPLICATIONS TO THE ANALYSIS OF INCOMPLETE PROGRAMS

Tese apresentada ao Programa de Pós--Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aluno: Leandro Terra Cunha Melo
Coorientador: Rodrigo Geraldo Ribeiro
Orientador: Fernando Magno Quintão Pereira

Belo Horizonte
April 2019

LEANDRO T. C. MELO

TYPE INFERENCE FOR C: APPLICATIONS TO THE ANALYSIS OF INCOMPLETE PROGRAMS

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais - Departamento de Ciência da Computação. in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Student: Leandro Terra Cunha Melo
Co-advisor: Rodrigo Geraldo Ribeiro
Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
April 2019

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Melo, Leandro Terra Cunha.

M528t Type inference for C: applications to the analysis of incomplete programs / Leandro Terra Cunha Melo — Belo Horizonte, 2019.
xxv, 158 p.: il.; 29cm.

Tese (doutorado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira
Coorientador: Rodrigo Geraldo Ribeiro

1. Computação - Teses. 2. Parsing. 3. Análise estática. 4. Source code generation. 5. Type inference. 6. Type theory. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*2.10(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

TYPE INFERENCE FOR C: APPLICATIONS TO THE ANALYSIS OF INCOMPLETE PROGRAMS

LEANDRO TERRA CUNHA MELO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
Prof. Fernando Magno Quintão Pereira - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]
Prof. Rodrigo Geraldo Ribeiro - Coorientador
Departamento de Computação - UFOP

[illegible]
Prof. Fábio Mascarenhas
Departamento de Ciência da Computação - UFRJ

Fernando J. Castor de Lima Filho
Prof. Fernando José Castor de Lima Filho
Centro de Informática - UFPE

[illegible]
Prof. Renato Antônio Celso Ferreira
Departamento de Ciência da Computação - UFMG

[illegible]
Prof. Rodolfo Sérgio Ferreira de Resende
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 24 de Maio de 2019.

Para Gusta con mucho Gusto.

Obrigado! Thank you! Danke!

Este trabalho tem sido uma jornada fenomenal. Ao longo desses últimos anos, cresci, fundamentalmente, como pessoa. Realizei descobertas, sem saber o que estava descobrindo. Tenho aprendido a aprender. Hoje, sinto-me mais capaz de entender o que eu não sei. Creio que essa seja uma sensação paradoxal inerente ao conhecimento: a cada novo saber, um novo universo de incertezas.

Várias pessoas contribuíram para minha formação como Doutor; e para o fruto material desse processo, que é esta tese. Direta ou indiretamente, toda contribuição da qual usufruí se enquadra em pelo menos um de dois grupos: naquele de contexto científico (*i.e.*, relacionado à linha de pesquisa em que atuei), ou no âmbito de todas as outras "coisas" da vida. Curiosamente, a conexão entre essas duas frentes pode ser bastante sutil. Em certos casos, impossível de ser desvinculada.

Meu primeiro agradecimento vai para a pessoa que, de fato, soube me orientar, **Fernando M. Q. Pereira**. O Fernando me apresentou às palavras e às ideias uma perspectiva matemática, com a qual eu não era familiarizado. Ele me ensinou a pensar e a me expressar no idioma da lógica; esteve presente e disponível em todas as etapas desse trajeto; enxergou à distância e guiou-me por uma trilha produtiva e segura; foi amigo e aberto para conversas sinceras (inclusive, para me incentivar a uma atividade paralela: a elaboração de um curso de C++). Muito obrigado, Fernando.

Através do Fernando, conheci o **Rodrigo G. Ribeiro**, quem veio a ser meu co-orientador. O Rodrigo produziu o embrião do que, eventualmente, se tornou o principal artefato de software deste trabalho. Foi durante esse momento inicial que eu comecei, sem que percebesse, uma espécie de libertação para a programação funcional, a qual me levou a um casamento com a teoria de tipos. Em inúmeras situações, o Rodrigo serviu como um ótimo consultor técnico. Muito obrigado, Rodrigo.

O que eu ainda não mencionei foi como cheguei até o Fernando. Lembro-me que estava sentado no chão de meu (já vazio) apartamento na Alemanha, a poucos dias de retornar definitivamente ao Brasil. Um amigo de infância, agora professor, me ligou para conversar sobre um projeto que coordenava. Ele, **Leonardo B. e**

Oliveira, procurava um engenheiro com meu perfil. Após algumas semanas, e sem muita ambição, lá estava eu de volta à escola. E foi assim... as portas de uma excelente academia de computação estavam abertas para mim. Muito obrigado, Léo.

A academia a qual me refiro é o **Departamento de Ciência da Computação** (DCC) da **Universidade Federal de Minas Gerais** (UFMG), uma instituição pública do **Brasil**. Nesse ambiente, tive a oportunidade de interagir com professores admiráveis e de debater com alunos inteligentes, sempre amparado por boa infraestrutura e uma equipe de funcionários eficientes. Em particular, no **Laboratório de Compiladores**, contei com um espaço confortável e propício ao estudo. Muito obrigado aos colegas do **Dê Drégons Néste**, especialmente ao **Marcus R. de Araújo**, e a todos que participaram da minha estadia no campus.

Deixo também meus agradecimentos aos diversos revisores que availiaram trechos desse trabalho contidos em múltiplas submissões a conferências científicas, fornecendo feedback relevante. Inclusive, aos professores **Fernando Castor Filho**, **Fábio Mascarenhas**, **Renato A. C. Ferreira**, e **Rodolfo S. F. de Resende**, os quais compuseram a banca da minha defesa de tese. Muito obrigado a todos vocês. A propósito, para que eu, antes de tudo isso, ingressasse no programa de doutorado, precisei de uma carta de recomendação. Muito obrigado **Renato C. Mesquita**, por tê-la oferecido.

Há ainda um professor que preciso mencionar. Porém, não por suas lições de matemática. Meu amigo, **Rodrigo T. N. Cardoso**. Muito obrigado por nossas conversas filosóficas, por sua riqueza espiritual e visão positiva do mundo, pelos passeios e companhia familiar e, é claro, pela motivação que me deu para encarar o doutorado.

Quem seria eu, se não fosse pelos maravilhosos pais que tenho... Foi ele, meu pai, **Fábio Souza Melo**, quem introduziu a mim (quando eu era um pequeno tenista) o significado de *garra*; de que, quando acreditamos e lutamos, a gente consegue. Sua dedicação ao trabalho, assim como seu empenho em "correr atrás" do que é necessário, inspiram o meu inconsciente – seu suor está presente a cada dia que vivo. Com meu pai, aprendi a ser correto. Incondicionalmente. Muito obrigado, pai. Tê-lo por perto é como sentir uma espécie de força oculta, dizendo que nada irá me faltar.

Oh, minha querida mãe, **Fátima Leal Terra Melo**! Quanta doçura, quanto amor, quanto cuidado... Quantas palavras de conforto, de sabedoria, de sensibilidade... De quanto se abdicou? Em tantas vezes me levantou; em quantas outras não me deixou cair? Em tantas vezes me fez rir; em quantas outras não me deixou chorar? Com quantos abraços me apertou? Com quantas mãos me segurou? Com quantos valores me educou? Tantos, tantos, tantos... Incontáveis. Muito obrigado, mãe. Seu carinho me ilumina; me abençoa; me protege; e me dá luz.

Falando em família, tenho já minha própria descendência. Meus dóceis e vibrantes, filha **Laura Mirachi Melo** e filho **Davi Mirachi Melo**. A energia de vocês é empolgante. A alegria de vocês é contagiante. Vocês não sabem, mas nossas brincadeiras e gargalhadas diárias funcionam como um combustível (aditivado) para minha alma. Não importa o quão pesado ou difícil é um dia de trabalho, chegar em casa e recebê-los em meus braços é uma bomba de alto astral. Muito obrigado, Chimbinha. Muito obrigado, Timburico. Vocês me enchem de orgulho e me dão o estimulo que preciso para continuar...

Minhas preciosidades não foram, obviamente, concebidas por mim, isoladamente. Elas foram gestadas com zelo único, pela minha inigualável esposa, **Carla Borim Mirachi Melo**. Se eu tive tranquilidade para me concentrar neste doutorado (e em minha carreira, como um todo), foi por eu saber que nossas crianças estavam, a qualquer instante, bem acolhidas e entretidas. Se eu, há cerca de 5 anos, mergulhei nesse mar, foi por que ela me deu um empurrãozinho e se dispôs e me ajudar a alcançar a margem. Se eu tive persistência para permanecer nesse barco, foi por que ela não permitiu que eu o deixasse virar. Muito obrigado, minha amável esposa; por me escutar (as palavras e por entre elas), por dialogar, por acomodar meu sofrimento, por compartilhar minhas emoções, pelo companheirismo.

Laura e Davi... crianças puras e lindas. Mas tão dependentes. Ainda que tenham uma mãe com super poderes, nossa rotina contou com o suporte da "brigada especial": os avós. Em relação a esse ponto, um segundo agradecimento aos meus pais. Além deles, muito obrigado aos meus sogros, **Silvana Borim Mirachi** e **Luiz Antônio Mirachi**; pelo leva e traz, pelo olha um e olha outro, pelo dorme na casa, pela carona para escola, e, naturalmente, pela atenção e carinho constantes.

Muito obrigado, meu irmão, **Renato Terra Cunha Melo**, pois nossas personalidades, reciprocamente influenciadas, estão entrelaçadas desde a origem. Muito obrigado aos meus próprios avós, que, já cedo, se preocupavam com minha educação e conjuntamente festejam minhas vitórias. Muito obrigado, **Milene de Mello**, pelos guias de sobrevivência e descobertas sobre mim mesmo.

Receio ser impossível devidamente agradecer a todos que me acompanharam por esse caminho. É até provável que eu não sequer saiba quem são exatamente todas essas pessoas. Caso o seu nome não esteja aqui, por favor aceite minhas desculpas.

Muito obrigado, meu Deus! Por tudo.

Resumo

A inferência de tipos é uma funcionalidade comum a diversas linguagens de programação. Enquanto que, no passado, ela era predominante em linguagens funcionais (*e.g.*, ML e Haskell), hoje em dia muitas linguagens orientadas a objeto ou multi-paradigmas tais como C# e C++ oferecem tal recurso. Ainda assim, a inferência de tipos para programas inteiros, não restrita apenas a expressões isoladas, continua sendo um problema em aberto no âmbito de C. A primeira dificuldade a ser driblada ao abordar esse problema é o fato da análise sintática dessa linguagem requerer, também, informações semânticas sobre programa. Além disso, inúmeros desafios complexos emergem ao longo do processo devido ao intricado sistema de tipos de C. Neste trabalho, apresentamos uma solução para este problema: uma abordagem baseada no algoritmo de unificação que nos permite inferir a estrutura de tipos da linguagem C.

Como principal aplicação de nossa técnica, investigamos a reconstrução de programas parciais. Código-fonte incompleto aparece naturalmente durante o desenvolvimento de software: nas etapas de projeto, evolução, testes, manutenção e, inclusive, visando a análise de programas. Portanto, a capacidade de entender fragmentos de código-fonte é um ativo valioso. Haja vista a variedade de tarefas nas quais tal conhecimento pode ser utilizado: para (i) habilitar ferramentas de análise estática em cenários onde componentes de software estejam inacessíveis; (ii) melhorar a precisão de ferramentas de análise estática que não exigem configurações extras/especiais; (iii) permitir que ferramentas para geração de testes e *stubs* possam ser aplicadas sem a necessidade de compilar todo um projeto; e (iv) auxiliar programadores na extração de estruturas de dados a partir de algoritmos. Nossa técnica foi avaliada em várias bibliotecas C, tais como o GNU Coreutils, a GNULib, o GLib do GNOME e a GDSL; em implementações disponíveis em um livro texto; e em trechos de código retirados de projetos como CPython, FreeBSD e Git.

Abstract

Type inference is a feature that is common to a variety of programming languages. While, in the past, it has been prominently present in functional languages (*e.g.*, ML and Haskell), today, many object-oriented/multi-paradigm languages like C# and C++ offer, to a certain extent, such a feature. Nevertheless, whole-program type inference is still an unsolved problem in C. The first difficulty encountered when tackling this problem is the fact that parsing C requires, not only syntactic, but also semantic information. Yet, greater challenges emerge due to C's intricate type system. In this work, we present a solution to this problem: a unification-based approach that lets us infer types that are not declared.

As a primary application of our technique, we investigate the reconstruction of partial C programs. Incomplete source code naturally appears in software development: during design, and while evolving, testing and analyzing programs. Therefore, the ability to understand it is a valuable asset. Reconstructing a partial program into a complete well typed one can: (i) enable static analysis tools in scenarios where components may be absent; (ii) improve precision of static analysis tools that require no build-specifications; (iii) allow stub-generation and testing tools to work on code snippets; and (iv) assist programmers on the extraction of data-structures from algorithms. We evaluate our technique on code from a variety of C libraries such as GNU's Coreutils, GNULib, GNOME's GLib, and GDSL; from implementations of a book; and on snippets from popular projects like CPython, FreeBSD, and Git.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Type inference, also known as type reconstruction, is a widespread feature of programming languages. In a program written with a language that offers such a feature, the declaration of a variable does not require an accompanying type annotation. The roots of type inference lie in the work of Hindley, Milner, and Damas [Hindley, 1969, Milner, 1978, Damas and Milner, 1982], and are supported by the unification algorithm of Robinson [Robinson, 1965]. This mechanism, which consists of the automatic deduction of the types of program terms by inspecting how they are *used*, has become popular through languages like ML and Haskell.

Nowadays, most programming languages provide certain level of type inference, but the capabilities offered by each of them vary a lot. Functional languages often offer a powerful inference mechanism that allows programs to be completely unannotated, even under the presence of high-order functions and different sorts of polymorphism [Cardelli and Wegner, 1985]. Yet another advanced type system appears in Scala. This language supports a mechanism that can infer the types of terms based on their *locality* in the program's *abstract syntax tree* (AST) [Pierce and Turner, 2000, Odersky et al., 2001]. Object-oriented and multi-paradigm languages like C# and C++ only provide a restricted form of type inference: when a variable is declared in conjunction with an initializer expression, a placeholder (`var` in C#, and `auto` in C++) may be employed to designated the type in question.

Given the ever-increasing popularity of type inference-enabled languages, we pose the following question: is it feasible to incorporate such a feature into the archaic type system of the (still popular) C language? The answer is yes. More precisely, the technique that we present in this work is capable of creating complete definitions for types that appear in a C program, which can be entirely free from `struct`, `union`, `enum`, or `typedef` declarations. Because our type inference is intended to work on existing code bases, we do not require a single syntax modification to the C language.

To illustrate our technique, let us present an example. The program in Figure 1.1 is a functional style implementation of the *mergesort* algorithm for linked lists. There are two primary data structures being used in that source code: (i) `node_t`, a recursive type that represents a node of the list - the field named `value` is responsible for carrying the element's data, the `next` field is a pointer to the following node; and (ii) `pair_t`, an auxiliary type that holds the sublists that are split at each iteration. Neither `node_t` nor `pair_t` are defined in the source. Therefore, if we attempt to compile our *mergesort* implementation with a standard C compiler, error diagnostics of the kind "unknown type" will be reported. However, by integrating type inference mechanisms, such as ours, as an early compilation phase, the following declarations for `node_t` and `pair_t` would be automatically produced.

```
struct TYPE_1__ { int value; struct TYPE_1__* next; };
typedef TYPE_1__* node_t;
struct TYPE_2__ { struct TYPE_1__* y; struct TYPE_1__* x; };
typedef TYPE_2__ pair_t;
```

The ability to write functional style C programs through the support of type inference is an achievement *per se*. Nevertheless, the principal application of our technique emerges in the context of partial programs. Incomplete source code appears in a variety of scenarios: during inception of a project, inside an editor or IDE; in cross-platform development, when certain source is unavailable due to incompatibilities; in the form of patches submitted for code reviewing; and as snippets contained in reports from bug-trackers. Therefore, the ability to understand 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 [Koppler, 1997, Knapen et al., 1999, Chugh et al., 2009, Dagenais and Hendren, 2008, Godefroid, 2014, Perelman et al., 2012].

In order to understand the source a partial C program, we need to deal with the problematic situation of encountering declarations that refer to types whose definitions might be missing. This setup corresponds quite accurately to that expected by our technique. Toward a solution to this problem, we present what, to the best of our knowledge, is the first *type inference* for C. With our system, partial program, $\mathcal{P}_p$, can be *reconstructed* into a new program $\mathcal{P}$. $\mathcal{P}$ preserves every syntax of $\mathcal{P}_p$, and includes any type declaration (possibly) missing from $\mathcal{P}_p$ that is necessary to make $\mathcal{P}$ well typed.

As a terminology note, we would like to clarify the meaning of the term "compilation" in this thesis. In the context of both C and C++, it is a common abuse

```
node_t merge_sort(node_t c) {
  if (!c) return 0;
  if (!(c->next)) return c;
  else {
    pair_t p = split(c);
    return merge(merge_sort(p.x), merge_sort(p.y));
  }
}

node_t merge(node_t a, node_t b) {
  if (!a) return b;
  else if (!b) return a;
  else {
    if (a->value < b->value) return new_node(a->value, merge(a->next, b));
    else return new_node(b->value, merge(a, b->next));
  }
}

node_t new_node(value_t value, node_t next) {
  node_t node = malloc(sizeof *node);
  node->next = next;
  node->value = value;
  return node;
}

pair_t split(node_t a) {
  pair_t s;
  if (!a) {
    s.x = 0;
    s.y = 0;
  } else if (!(a->next)) {
    s.x = a;
    s.y = 0;
  } else {
    pair_t p = split(a->next->next);
    s.x = new_node(a->value, p.x);
    s.y = new_node(a->next->value, p.y);
  }
  return s;
}
```

Figure 1.1. Functional style implementation of the *mergesort* algorithm for linked lists. If the content of this figure is pasted, *as is*, into a file and an attempt to compile it with an standard C compiler is made, error diagnostics of the kind "unknown type" will be reported. These errors can be explained by the absence of declarations for the types `node_t`, `pair_t`, and `value_t`. However, if our type inference mechanism is integrated to the compilation process, suitable definitions for `node_t` and `pair_t`, along with an synonym declaration to `value_t`, would be automatically produced and, consequently, a successful compilation is obtained.

of wording to refer to compilation as the entire pipeline of *building* a program: (a) preprocessing, (b) compiling, (c) assembling, and (d) linking. Our work is specifically targeted at compilation, as in item (b). For instance, we do not generate function definitions (*i.e.*, bodies for functions). Consequently, a program reconstructed with our technique may not link. Nevertheless, stub-generation tools exist [Cadar et al., 2008, Godefroid et al., 2005, Williams et al., 2005, Tillmann and De Halleux, 2008]. Those are, however, beyond the scope of this work.

1.1 Discovering the Intricate Type System of C

A C program, in its usual format, does not require that its types are inferred. After all, in this language it is expected that a declaration comes annotated with a type, and that the definition of such type is available in the program. In spite of that, there is a variety of scenarios where the definition of a type may be absent - we study those in Section 5.3. Upon this situation, combined with an eventual need of, nevertheless, working with such a program, the only possible solution is to "discover" what the missing types look like. To this end, a type inference system must be employed.

In order to produce a well typed C program out of an incomplete source code (the meaning of "incomplete" is formalized both in Chapter 2 and 3, just slightly differently), we have to circumvent a number of challenges, which we now exemplify. Even though this selection of snippets reflects the predominant difficulties faced by our technique, inferring types that satisfy C's type system involves additional hurdles. Those are not necessarily mentioned in this section but they appear throughout the text. We start with Challenge 1, which concerns the parsing of a C program.

Challenge 1. Determine the syntactic nature of program identifiers in a language that relies on semantic information to guide parsing, considering that not all declarations may be present.

Figure 1.2 illustrates this challenge. Program (a) does not contain enough information to determine the syntactic nature of `T`. We could infer it either as an arithmetic type[1] or as a variable. The former hypothesis corresponds to the program in Figure 1.2 (b); the latter, to the program in Figure 1.2 (c). In Section 2, we discuss how to postpone the decision about the nature of `T` until the inspection of additional program syntax can provide us with disambiguation information. For instance, in Figure 1.2 (d), the declaration `T b;` allows us to deduce that `T` is a type; a situation where an analogous conclusion indicates that `T` is a variable appears in the programs (e) and (f) of this Figure 1.2. There, `T` participates as an operand of a binary expression.

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

(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;
}
```

Figure 1.2. (a) In this program, is `T` the name of a variable or of a type? (b) A program where we know that `T` is a type because a declaration for it exists. (c) A program where `T` is declared as a variable. (d) Even though the program does not contain a declaration for `T`, we can conclude that `T` is a type (the grayed out part is what could be a valid declaration for such type). (e-f) Programs where, through the aid of extra syntax, it is possible to conclude that `T` is a variable.

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;
}
```

(d)
```
void f() {
  T d;
  d + u;
}
```

(e)
```
void f() {
  T e;
  e + p;
  *p = 10;
}
```

(f)
```
void f() {
  T e;
  p + e;
  *p = 10;
}
```

Figure 1.3. (a) In this program, is `T` an arithmetic or a pointer type? (b) A program where `T` is an arithmetic type, since pointers cannot participate in a division operation. (c) A program where `T` is a pointer: the only type that supports the dereferencing expression. (d) Once more, is `T` an arithmetic or a pointer type? (e) Not only `T` must be an arithmetic type, but it is restricted to an integral one, *e.g.* an `int`. (f) Because the order of operands is not important in an addition, `T` is again an integral type.

Neither the program in Figure 1.3 (a) nor the one in Figure 1.3 (d) are syntactically ambiguous. In both cases, it is clear that `T` must be a type. However, those programs are semantically ambiguous. In the former case, `T` can be `int`, `int*`, `int**`, `long`, `long*`, `float`, `double**`, etc. In the latter, `T` can be either an arithmetic or a pointer type, given that, in C, there are two possibilities for the operands involved in an additive expression: (i) both of them have arithmetic types; or (ii) one of them is a pointer, and the other an arithmetic type – further restricted to be an integral type.

As we explain in Section 3, we devise a lattice of *shapes* that helps us find *most general types* through the analysis of extra program syntax. This lattice lets us promote `T` to *numeric* in Figure 1.3 (b), and to *pointer* in Figure 1.3 (c): these choices are justified by the fact that pointers cannot be used in a division operation [ISO-Standard, 2011]{§6.5.5}, and that C does not permit dereferencing arithmetic types. In a similar way, our lattice classifies, in Figure 1.3 (e), `p` as a *pointer* and `e` as an *integral*. The conclusions that can be drawn for Figure 1.3 (f) are the same as those from Figure 1.3 (e), since the order of operands in addition does not affect typing.

Challenge 3. Account for unidirectional type relations 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;
}
```

(e)
```
void f() {
  T5 h;
  double m = h;
}
```

(f)
```
void f() {
  double m;
  T6 h = m;
}
```

Figure 1.4. (a) A program where `T1` can be of the same type of `x`, or also of a `const`-qualified compatible integral, *e.g.*, `const int`. (b) In this program, `T2` cannot be inferred as `const int`, even though that is the exact type of `y`. (c) Again, a situation where, despite the presence of an assignment, `T3` cannot be `const int*` – although it could be `int*`. (d) Can the type `T4` be `int*` in this program? If not, for what reason? (e) A program where `T5` could be `double`, `int`, `short`, etc. (f) Can the type of `T6` be `int`? If not, for what reason?

This challenge exists due to the implicit conversions that are allowed in C. In particular, those involving *type qualifiers* [ISO-Standard, 2011]{§6.7.3}. Even though we focus on `const`, the described behavior applies to `volatile` as well. In Figure 1.4 (a), `T1` can be an `int` or `const int` (or any other compatible integral type, such as `long`). The latter is possible because a *constant* variable may be created out of a non-constant one. But, in Figure 1.4 (b), `T2` must be `int` (or another non-`const` integral type), because typing `T2` as `const int` would lead to an invalid program, since assigning to a constant variable, as in `b = 10`, is not permitted – constants can only be *initialized*.

The challenge gets harder in Figure 1.4 (c) due to the presence of pointers. Although `w` has type `const int*`, it is legal to have it assigned by a type with weaker qualification [ISO-Standard, 2011]{§6.3.2.3-2}, like `int*`. In fact, the expression `*c = 10` indicates that `c` cannot be a constant, and the correct solution is to have `T3` as `int*`. However, in Figure 1.4 (d) we have an assignment in the other way around. In this case, the constant pointer appears on the right-hand-side, and `T4` cannot be `int*`, since that would mean a break in the promise of immutability; making `T4 const int*`, on the other hand, would be correct.

Another domain of implicit conversions existing in C is the one relating to arithmetic types. In this context, we are specifically interested on avoiding the *truncation* of values [ISO-Standard, 2011]{§6.3.1.4-1}. The programs in Figure 1.4 (e) and (f) illustrate this situation. In the former case, we could infer the type of `T5` as an `int` or a `double`. On the other hand, such choice is not allowed for the latter case: typing `T6` as `int` could lead to the loss of data. This fact is a potential source of *undefined behavior*. Due to the aforementioned asymmetries, classical unification, which relies on type equivalences, cannot be used in C. In Section 4, we discuss how to employ subtyping for solving this problem, together with a novel unification approach.

Challenge 4. Generate types for variables whose nature is not 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;
}
```

Figure 1.5. What are the types `T1`, `T2`, and `T3`, considering that `malloc`'s return type is `void*`?

Figure 1.5 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 `void*`. And yet, we need to instantiate it to produce a well typed program for the incomplete source in Figure 1.5 (c). Further syntax in Figures 1.5 (a-b) lets us conclude that `T1` and `T2` are arithmetic types; on the other hand, in Figure 1.5 (c) we do not have this information. In Section 4.2 we define the notion of an "orphan"; `T3` is such one. In particular, it can be safely instantiated to an arbitrary type, as long as it is a pointer.

1.2 The Contributions of This Work

The thesis of this work is that it is possible to implement a type inference mechanism for the C programming language. During the course of designing such a technique, we have architected solutions to challenging aspects of C's type system. Through the mechanism that we developed, a number of tools can be leveraged to better function on a setup where source code is incomplete. The list below describes what we believe to be our main contributions.

- Parsing C requires semantic knowledge in order to handle ambiguous syntax. However, such information might not be available if a program lacks just a single type declaration. In Section 2 we formalize a technique that deals with this problem: parsing decisions are postponed until further syntax can be extracted from the source code.
- The C language accepts liberal conversions. In particular, 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 cannot be unified via a classical algorithm, as typically employed in type inference. In Section 3 we explain how to discover the nature of a type by means of a lattice of shapes (*e.g.* a pointer or an arithmetic shape). Then, we use this information to prevent inconsistent unifications.
- In C, implicit conversions of qualified (*e.g.* `const` and `volatile`) and pointer types are asymmetric. This fact prevents us from using standard inference techniques, since they rely on type equivalences. In Section 4 we discuss a strategy to model pointer relations through subtyping, and an enhanced unification algorithm that is capable of simultaneously dealing with equivalence and inequality constraints.
- A program might not contain enough uses of its variables and functions so that all types can be inferred. In Section 4.2 we describe how our type inference works in this scenario. Unsolved types can be safely instantiated, without interfering with solved type variables.
- To demonstrate the ideas advocated in this work, we have materialized them into a tool called **PsycheC**[2], presented in Section 5. This tool, which is a practical contribution of this work, produces a C header containing any declaration that is absent from the program that it receives as input. The `#inclusion` of this

[2]https://github.com/ltcmelo/psychec

header in the source code characterizes a *reconstructed* program that is expected to compile successfully.

- In Section 5.3, we discuss how the successful reconstruction of a program enables static analysis tools to work, even under a restricted environment; improves the precision of "zero setup" static analysis tools; supports testing and stub-generation; and serves as a general code completer for C programmers.
- We provide, in Appendix A, a full implementation of the μC language that we use in Section 3.1 to explain the core ideas of our type inference. Given the subtleties involved in our technique, and the fact that it combines a variety of topics from the literature, we believe that an implementation that accurately follows, side-by-side, a given formalism has a relevant pedagogical value.

1.3 A Brief Roadmap for This Thesis

The type inference mechanism that we propose consists of the following parts, in the sequence as they are here enumerated.

1. A program, $\mathcal{P}$, that lacks the declarations of types may contain syntax ambiguities. Therefore, we must first parse the source code and arrive at an *abstract syntax tree* (AST) that is unambiguous. This is the topic of Chapter 2.
2. To express the typing relations existing in C, we create a constraints language. Such constraints are generated by means of traversing the AST of $\mathcal{P}$. Chapter 3 contains the details of this process.
3. Solving the constraints of $\mathcal{P}$ requires a solver that is capable of simultaneously dealing with both type equivalence and subtyping. We elaborate one in Chapter 4. A complete solved form of our system let us infer the types that are necessary to make $\mathcal{P}$ well typed.

Throughout the text, three supporting languages are introduced: μA, μB, and μC. All of them are a subset of C, but contain only enough syntax that is essential to the illustration of our ideas. Among them, μC is the principal one: it is used as the basis of our formalism; a Haskell implementation of it is available in Appendix A. Nevertheless, μC is not capable of dealing with C programs. To this latter end, we have developed PsycheC, an industrial-strength tool that, alongside with an empirical evaluation, is presented in Chapter 5. Related work appear in Chapter 6 and final remarks in Chapter 7.

Chapter 2

Parsing Ambiguous Syntax

The C language uses a *symbol table* to guide parsing. Specifically, a parser checks the content of this table to determine which grammar production is to be accepted, depending on whether an identifier designates a program variable or the name of a type. If declarations were not mandatory, it would be impractical to rely on such semantic information. Therefore, when designing a broadly comprehensive type inference technique, the syntactic analysis that we perform must be still precise, yet flexible enough to deal with the absence of declarations.

Revisiting Challenge 1

In Figure 1.2, from Section 1.1, we show programs with ambiguous syntax. All of them contain the term `x * y;`. This construct can be interpreted as the multiplication of variables `x` and `y`, or the declaration of variable `y` as a pointer to a type named `x`. When possible (*i.e.*, the source is not inherently ambiguous), our parser must be able to disambiguate such terms.

A known approach for parsing incomplete or erroneous syntax may is *fuzzy parsing* [Koppler, 1997]. Yet, a fuzzy parser, by definition, does not rigorously recognize a language. Such imperfection poses difficulties to the development of a type inference mechanism, since important information may be lost. To overcome this situation, Knapen *et al* [Knapen et al., 1999] present an alternative approach, only resembling fuzzy parsing, to deal with this problem. In their work, eight ambiguous constructs of the C++ language are addressed. Out of those, three exist in C as well. We show them in Table 2.1.

Table 2.1. Ambiguities that can happen due to missing declarations. The first row of the table shows a syntax that is typical of a function call. However, if `a` is the name of a type, that syntax would actually denote the declaration of a variable `b` (the parenthesis are ignored). In the second row, when `a` is a type, we have a cast expression for both constructs (the value dereferenced by `*b` and of `-b`), instead of multiplication and subtraction, as they might appear. The last row is the classical pointer versus multiplication ambiguity.

Ambiguity	**Syntax**
Function call *or* variable declaration	`a(b);`
Cast of unary expression *or* binary expression	`(a)*b; (a)-b;`
Pointer declaration *or* multiplication	`a * b`;

2.1 Properties of an Ambiguous-Program AST

Our syntax disambiguation strategy is based on Knapen's, but with additional formal guarantees. The principle behind a parser that can deal with programs where declarations may be missing is to postpone certain decisions until there is enough information to conduct disambiguation. To explain this strategy, we shall use a language called μA, whose syntax appears in Figure 2.1. μA offers the minimum setup that allows us to build semantically different versions of `a * b;`. While we do not describe all the ambiguity cases in Table 2.1, their handling is similar to the one we now illustrate.

The syntax of μA follows immediately from a corresponding the Prolog *Definite Clause Grammar* (DCG) [Sterling, 1994, Ch.16] appearing in Figure 2.2. There, a program is a list of *terms* separated by a semicolon, `;`. Terms may be comprise of a type declaration, *Td*, a variable declaration, *Vd*, or an expression, *E*. The top-level non-terminal, P, has four attributes, T_i, V_i, T_s, and V_s: the first two are *inherited*; the other two are *synthesized*. Thus, production P receives T_i and V_i and, if it successfully consumes a string, it produces T_s and V_s.

We let T denote sets of names used as types, and V denote sets of names used as variables. Terms, which are denoted by S, use the same four attributes T_i, V_i, T_s, and V_s. A type declaration $Td(x)$ succeeds if x is found to be the name of a new type - only the type `int` is builtin in μA; hence, new types are aliases 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. A *valid* μA program is formalized by Definition 1.

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

Figure 2.1. The μA language. This is a minimalistic language that contains an ambiguous syntax that mimics the pointer declaration versus multiplication ambiguity of C. We assume that the only valid identifiers of μA are x, y, and z. Note that the term $x * y$ can be derived by either Vd or E.

Definition 1 (Valid μA Program)**.** We say that $\mathcal{P}$ is a valid μA program if $\mathcal{P}$ is a *list* of terms Td, Vd, or E, that can be derived from production $P(\varnothing,\varnothing,T,V)$ using the grammar in Figure 2.1. In this case, we say that T is the set of *type names*, and V is the set of *variable names* of $\mathcal{P}$. $\diamond$

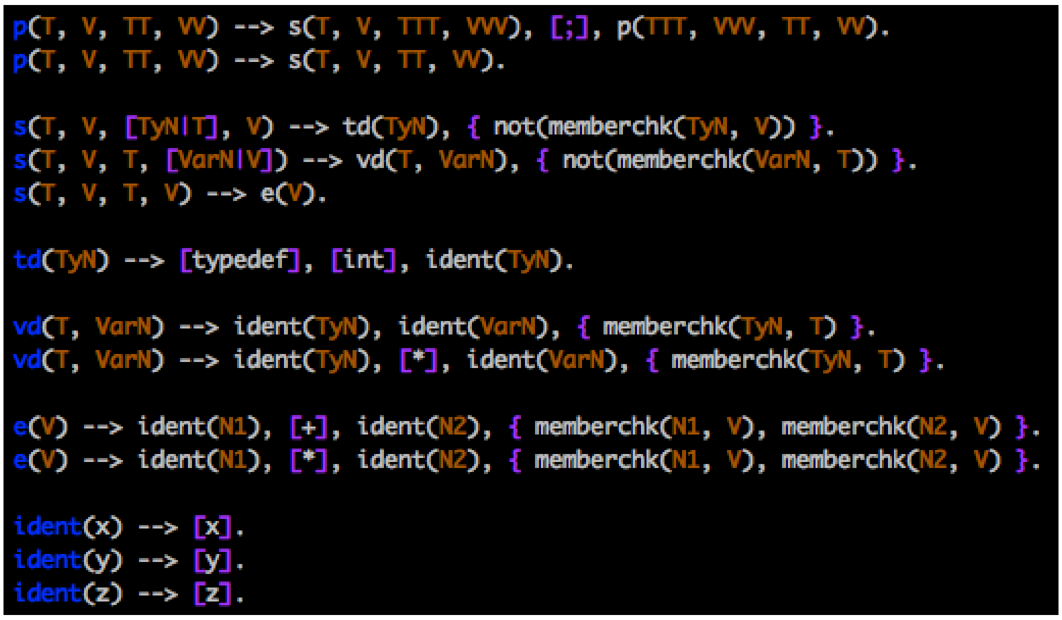
Lemma 1 (Properties of a Valid μA Program)**.** *The following properties are true about a valid program $\mathcal{P}$. Below, ++ is the list concatenation operator.*

1. $T \cap V = \varnothing$.
2. *If* $\mathcal{P} = \mathcal{P}_1$ ++ `a b;` ++ $\mathcal{P}_2$*, then* $\exists x \in T$*, such that* `typedef x a;` $\in \mathcal{P}_1$.
3. *If* $\mathcal{P} = \mathcal{P}_1$ ++ `c + d;` ++ $\mathcal{P}_2$*, then:*
   (i) $\exists x \in T$*, such that* `x c;` $\in \mathcal{P}_1$*,*
   (ii) $\exists y \in T$*, such that* `y d;` $\in \mathcal{P}_1$*, and*
   (iii) $\{\texttt{c}, \texttt{d}\} \subseteq V$.

Proof. Direct from the grammar productions of μA.

1. Follows from the conditions in productions 3 and 4, *i.e.*, $x \notin V$ and $x \notin T$.
2. Follows from the condition in production 7, combined with that of 3 and 6.
3. Similar to the proof of 2.

$\square$



```
p(T, V, TT, VV) --> s(T, V, TTT, VVV), [;], p(TTT, VVV, TT, VV).
p(T, V, TT, VV) --> s(T, V, TT, VV).

s(T, V, [TyN|T], V) --> td(TyN), { not(memberchk(TyN, V)) }.
s(T, V, T, [VarN|V]) --> vd(T, VarN), { not(memberchk(VarN, T)) }.
s(T, V, T, V) --> e(V).

td(TyN) --> [typedef], [int], ident(TyN).

vd(T, VarN) --> ident(TyN), ident(VarN), { memberchk(TyN, T) }.
vd(T, VarN) --> ident(TyN), [*], ident(VarN), { memberchk(TyN, T) }.

e(V) --> ident(N1), [+], ident(N2), { memberchk(N1, V), memberchk(N2, V) }.
e(V) --> ident(N1), [*], ident(N2), { memberchk(N1, V), memberchk(N2, V) }.

ident(x) --> [x].
ident(y) --> [y].
ident(z) --> [z].
```

Figure 2.2. A Prolog implementation of the μA language. If we provide `p([], [], T, V, [typedef, int, z, ;, z, *, y, ;, z, x, ;, x, *, y], [])` as an input to this program, the result obtained is composed by sets `T = [z]` and `V = [x, y]`. This means that `z` is a type name, while `x` and `y` are variable names.

Example 1. Program $\mathcal{P}_1$ = `typedef int a; a a;` is not valid, because `a` is used both as the name of a variable and of a type. Program $\mathcal{P}_2$ = `a b;` is not valid either, because the name `a` is used as a type, but it has not been previously defined by a `typedef`. Program $\mathcal{P}_3$ = `int a; a + b;` is not valid, since the name `b` is used as a variable, but it has not been declared.

A valid μA program does not contain ambiguities. The term `a * b` can always be disambiguated by inspecting the sets T and V for the presence of `a`. The same situation happens in C, since disambiguation of such syntax would be done by looking up the symbol table and checking whether `a` is the name of a type or the name of a program variable. Therefore, in order to reproduce the scenario desired for our type inference, where type declarations are not necessarily present, we need to emulate the absence of the semantic information that would be available to a compiler under normal circumstances. To this end, Definition 2 formalizes the notion of a *partial program*.

Definition 2 (Partial μA Program)**.** Let $\mathcal{P}$ be a valid μA program. We obtain a partial program, $\mathcal{P}_p$, by eliminating any number of terms from $\mathcal{P}$. ◇

Example 2. Program $\mathcal{P}$ = `typedef int a; a b; a * c;` is valid, according to definition 1. There exist eight possible partial programs that we can produce out of $\mathcal{P}$. A few of them are: $\mathcal{P}_1$ = `a b; a * c;` , $\mathcal{P}_2$ = `typedef int a; a * c;` , and $\mathcal{P}_3$ = `a b; a * c;` .

It can be noticed in Example 2 that partial programs are not always valid; to the contrary, the process of eliminating terms is likely to produce an invalid program. Furthermore, such partial programs may contain an ambiguity just like the one that would exist in C, if type declarations are absent. For instance, the partial program $\mathcal{P}_p$ = `a * b;` is ambiguous because we do not know if `a * b` is a multiplication between `a` and `b`, or the declaration of `b` as a pointer to a value of type `a`. Nevertheless, there are partial programs that, despite being invalid, provide us with enough information for disambiguation. Consider $\mathcal{P}_p$ = `a * b; a c;` . In this case, we can only have two declarations: the second term, `a c`, lets us infer that the name `a` must be the name of a type, not that of a program variable. Analogously, the two terms in $\mathcal{P}_p$ = `a * b; a + c;` must be expressions, for the second one lets us infer that `a` is the name of a variable.

Based on the observation that certain program terms let us disambiguate the ambiguous ones, we define the language μB, whose logical grammar appears in Figure 2.3 - and Prolog implementation in Figure 2.4. This grammar uses a new non-terminal Abg to carry over an ambiguity: upon matching `a * b` we postpone the decision of whether `a` is a type or variable. This name, `a`, will only be marked as a type if the partial program contains either a term like `typedef int a`, or a declaration such as `a b`. Similarly, we mark `a` as a variable if the partial program contains either a declaration such as `x a`, or an expression like `x + a`. If all ambiguities in a partial program can be resolved, we have a *unambiguous program.*

Definition 3 (Unambiguous Program)**.** Let $\mathcal{P}_p$ be a partial program. We can successfully disambiguate $\mathcal{P}_p$ if the production $P_p(\varnothing, \varnothing, T, V)$ succeeds on $\mathcal{P}_p$, and every name in $\mathcal{P}_p$ is either in T or in V. In this case, we refer to $\mathcal{P}_p$ as a *unambiguous program.* ◇

Despite the fact that an ambiguous partial program can be successfully parsed by the grammar of Figure 2.3, it may happen that such program cannot be disambiguated. That is the reason why we, in Definition 3, require that every name in the program is put into either T or V, *i.e.* it is *observed* as a type or as a variable, respectively. As an example of the inability to determine the nature of a name, in $\mathcal{P}_p$ = `int x; a * x;` no verdict about `a` can be made. Notwithstanding, we can prove several properties of partial programs, even if they are still ambiguous, as we state in Theorem 1. This theorem, which allows us to correlate $\mathcal{P}$ and $\mathcal{P}_p$, gives us Corollary 1.

$$
\begin{array}{lll}
P_p(T, V, T', V') & ::=_a & S_p(T, V, T'', V'');\ P_p(T'', V'', T', V'); \\
P_p(T, V, T', V') & ::=_b & S_p(T, V, T', V'); \\
 & & \\
S_p(T, V, T, V \cup \{y\}) & ::=_c & Abg_p(y) \\
S_p(T, V, T \cup \{z\}, V) & ::=_d & Td_p(z) \\
S_p(T, V, T \cup \{z\}, V \cup \{x\}) & ::=_e & Vd_p(z, x) \\
S_p(T, V, T, V \cup \{x, y\}) & ::=_f & E_p(x, y) \\
 & & \\
Abg_p(y) & ::=_g & x * y \\
 & & \\
Td_p(z) & ::=_h & \texttt{typedef int}\ z \\
 & & \\
Vd_p(z, y) & ::=_i & z\ y \\
 & & \\
E_p(x, y) & ::=_j & x + y
\end{array}
$$

Figure 2.3. The μB language. This is the grammar we use to disambiguate programs. Instead of having two different productions that derive $x * y$, only one is provided, Abg_p. On the other hand, upon matching of this syntax, we cannot associate a neither to T nor to V. Such decision is postponed until sufficient information is available to determine whether `a` is a variable or a type.

Theorem 1 (Correspondence between Partial and Original Programs)**.** *Let $\mathcal{P}_p$ be a partial program of $\mathcal{P}$. If $P_p(\varnothing, \varnothing, T_p, V_p)$ succeeds on $\mathcal{P}_p$, and $P(\varnothing, \varnothing, T, V)$ succeeds on $\mathcal{P}$, then the following properties hold:*

1. If $x \in T_p$, then $x \in T$.

2. If $x \in V_p$, then $x \in V$.

Proof. Direct from the grammar productions of μB.

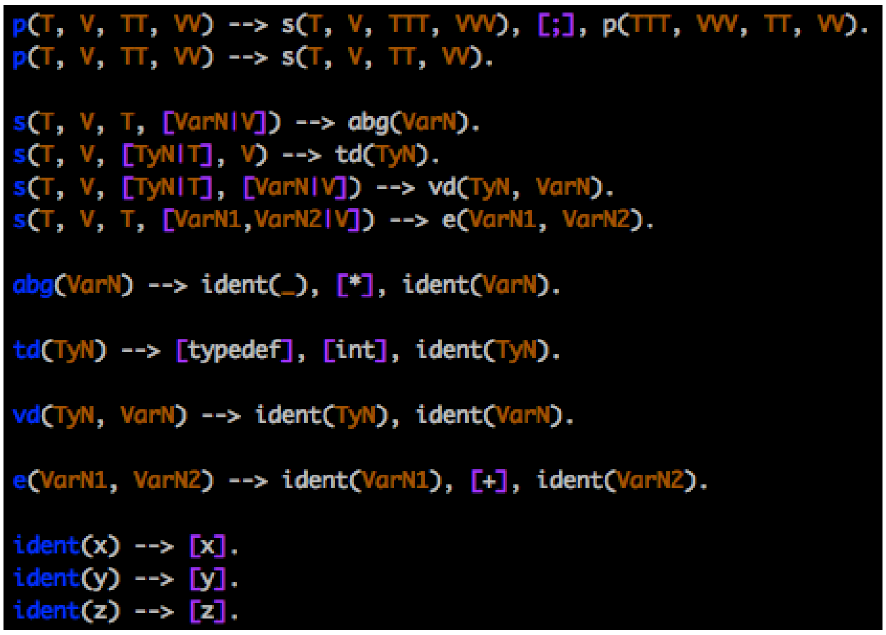
1. Only productions d and e can insert names into T_p. Production e corresponds to production 6, from Figure 2.1; and e to production 7. While the productions of μA contain checks, *e.g.* requiring x to be in T, those in μB contain a side-effect, *i.e.* they insert x into T_p.

2. Productions c, e, and f insert names into V_p. If $x \in V_p$ due to f, then x is the second operand of an addition – the check in production *9*, from Figure 2.1 requires $x \in V$; if $x \in V_p$ due to e, then x appears in a term such as `a x`. Hence, productions *4* and *7* insert x into V; finally, if $x \in V_p$ due to c, then $\mathcal{P}_p$ contains a term `a` $*$ `x`. This term can be parsed in μA by either productions 8 or 10. Both contain guards ensuring that x is a variable.

$\square$

Corollary 1 (Empty Intersection of Type and Variable Sets)**.** *Let $\mathcal{P}_p$ be a partial program. If $P_p(\varnothing, \varnothing, T_p, V_p)$ succeeds on $\mathcal{P}_p$, then $T_p \cap V_p = \varnothing$.*

Proof. Follows from Theorem 1 and Lemma 1, Property 1, which ensures that names in the program denote either types or variables. □



```
p(T, V, TT, VV) --> s(T, V, TTT, VVV), [;], p(TTT, VVV, TT, VV).
p(T, V, TT, VV) --> s(T, V, TT, VV).

s(T, V, T, [VarN|V]) --> abg(VarN).
s(T, V, [TyN|T], V) --> td(TyN).
s(T, V, [TyN|T], [VarN|V]) --> vd(TyN, VarN).
s(T, V, T, [VarN1,VarN2|V]) --> e(VarN1, VarN2).

abg(VarN) --> ident(_), [*], ident(VarN).

td(TyN) --> [typedef], [int], ident(TyN).

vd(TyN, VarN) --> ident(TyN), ident(VarN).

e(VarN1, VarN2) --> ident(VarN1), [+], ident(VarN2).

ident(x) --> [x].
ident(y) --> [y].
ident(z) --> [z].
```

Figure 2.4. A Prolog implementation of the μB language. If we provide `p([], [], T, V, [z, *, y, ;, z, x], [])` as an input to this program, the result obtained is composed by sets `T = [z]` and `V = [y]`. Because `z` is an element of the set of types, we can disambiguate `z * y` as a declaration.

2.2 From an Unambiguous AST Onwards

Programs whose AST can be disambiguated are eligible for type inference. While we have, in Section 2.1, drawn a correspondence between a supposedly partial and original program, such relation is not important for type inference itself. Regardless of how the variable and type names can be associated in any two programs, as long as our parser is able to produce a correct AST, the technique that we propose can always be employed. This is what we now explain in Chapter 3.

Chapter 3

The μC Language and Constraints

In C, a programmer is allowed to define new types through the declaration of an `enum`, `struct`, or `union`. Moreover, the `typedef` construct can be used to declare a type synonym. In a typical inference setup, the type of program terms would be deduced in conformity to such declarations and to those of builtins, *e.g.*, `int` and `double`. But if declarations are not available, then a type inference-enabled language must be able to synthesize complete definitions for the type used in a program. However, not only due to syntax ambiguities but to semantic ones too, standard type inference alone is not enough to achieve this task.

Revisiting Challenge 2

In Figure 1.3, from Section 1.1, we show programs with ambiguous meaning. For instance, consider (a) and (d). In the former case, an ambiguity exists because the constant `0` can be assigned either to an arithmetic or pointer type; in the latter case, it is just as possible that `T` refers to an arithmetic or pointer type too. But, if `u` is a pointer in program (d), then `T` must be an arithmetic type that is an integral. There are other constructs in C that lead to similar ambiguity problems. For instance, the expression `{1, 2, 3, 4}` can be used to initialize different aggregate types[1]: an array of four integers values (`int[4]`), a `struct` with four integer fields (`struct T {int a, b, c, d;}`), or an array of two `struct`s, each with two integers [ISO-Standard, 2011]{§6.7.8}.

The type inference mechanism that we propose adheres to standard practice: it

[1]An aggregate type collectively refers to arrays and structures [ISO-Standard, 2011]{§6.2.5.21}.

comprises a constraint generation phase, and a subsequent phase that solves those constraints [Rémy, 2017, Ch.5]. But to deal with the type ambiguities of C, we summarize, prior to constraint generation, information about how expressions[2] are *use*d throughout the program. Based on such information, we build a lattice of *shapes*. In this lattice, every AST expression is bound to a shape. During this preliminary phase, we look for syntax that lets us move expressions up in the lattice until a fixed-point is reached - this action corresponds to the disambiguating strategy we employ for the μB language, as described in Chapter 2. The details about our lattice for arithmetic and pointer types are presented in Section 3.4; a lattice for dealing with aggregate types can be built under the same principles.

Before we move forward, let us discuss a few more features of C. In this language, programmers can further refine a type by attaching to it a *type qualifier* [ISO-Standard, 2011]{§6.7.3}, such as `const` or `volatile`[3]. These qualifiers cause certain relations among types to be *unidirectional*, incapacitating classical unification, an algorithm that relies on symmetric relations between types, as a solving procedure. This asymmetry that qualifiers impose on type relations can be particularly well seen upon assignments.

Revisiting Challenge 3

In Figure 1.4, from Section 1.1, we show programs that illustrates asymmetric type relations. In program (a), it is possible to type `T1` as either `int` or `const int`[4]. At first sight, this choice exists in Figure 1.4 (b) too. However, because a `const` variable cannot be modified – it could have been *initialized*, though – inferring `T2` as `const int` yields an illegal program, for the promise of immutability imposed on `b` is broken with `b = 10`.

Forgetting about `const` is a convenience also allowed in program (c). But now, the situation is the opposite, and the types in question are pointers! Here, it is possible to discard `const` because an implicit conversion from a non-qualified to a qualified pointer is permitted: the latter denotes a type more restricted than the former [ISO-Standard, 2011]{§6.3.2.3-2}. In fact, typing `T3` as `int` would be incorrect, given the expression `*c = 10`.

[2]Type specifiers within declarations are accounted as well, in a similar (and simpler) manner. But we restrict the presentation to expressions.

[3]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. Yet, both qualifiers share typing rules, with `const` being stricter - without account of dynamic semantics

[4]Formal parameters and arguments of functions would be addressed in a similar manner: `T` must be

Finally, let us consider program (d). 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. Otherwise, we would once more break a promise of immutability.

Based on the facts that we have observed, two conclusions can be drawn about the typing relations of C. Whenever there is an assignment (or a similar binary operation) between non-pointer types, it is always safe to discard any qualifier from the types in either side of the expression. While the absence of `const` or `volatile` might come as an expressivity loss from the standpoint of readability, it never renders a program invalid in the perspective of type checking. We call such dropping of a qualifier as the *qualifier-neutral* strategy.

The second conclusion that we can draw is subtler than the first one. Relations involving pointers must be treated with additional care. Since the same memory location would be accessible by different objects, in order to ensure that any restriction, such as the promise of immutability, is not broken through a dereferencing expression, type qualifiers must be taken into account. In particular, a `const` or `volatile` that appears on the right-hand-side of an assignment must be propagated to the left-hand-side. We call this the *qualifier-aware* strategy.

The last aspect of C that we bring to attention is that, when type declarations are missing, syntax might not be sufficient to restrict at all the type of a variable. In reality, the same syntax can be associated to an infinite variety of types (*i.e.* unbounded polymorphism). That classical example of this situation appears with `void*`. This is an opaque type that is intended to be used only during conversions; there exists no `void` data, and dereferencing such a pointer is not permitted. What if a program has variables which are neither initialized nor used?

Revisiting Challenge 4

In Figure 1.5, from Section 1.1, we show programs where a same type, the return of function `malloc` has different instantiations. In program (c), it is not possible to determine what could have been the original `T3`. Actually, all it takes to make that a valid program is to provide a declaration such as `typedef int T3` (any other concrete type would fit the purpose).

`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 the formal parameters correspond to its left-hand-side.

We have, in the previous paragraphs, initiated an exploration of what we consider to be the most interesting challenges to the design of a type inference-enabled variation of C. Those are not the only difficulties that one would face, however. But since formalizing our technique for C in its entirety would be too laborious of a task – yet, with parts without significant scientific interest, we proceed with a subset of the language. Further characteristics of C's type system are discussed in Chapter 5.

3.1 The Definition of μC

For the explanation of our type inference technique, we define a new language that, just like C, contains the semantic ambiguity between a pointer and an integral. Such language, which we call μC, also has other features like a *nominal* type system [Pierce, 2004, Ch.19] and qualified types, so that it is rich enough to allow us to explain the intricacies of the C type system. From now onwards, our presentation can be mapped, side-by-side, to a Haskell implementation of μC that is available in Appendix A and online[5]. Nevertheless, we call the attention of the reader to the following fact: the *raison d'être* of μC and its accompanying implementation is to support our theoretical discussion; in order to perform type inference on a C program [ISO-Standard, 2011], PsycheC, which we introduce in Section 5, is the tool to be used. Whenever it is appropriate, we correlate μC with PsycheC.

The syntax of μC appears in Figure 3.1; its implementation is available in Appendix A.1. We use a *star* superscript, *, for the Kleene closure of grammar elements, and an *asterisk*, $*$, for pointer-related constructs. P stands for a program, F for a function definition, D and D_s for declarations. A declaration D can be of a program variable, a formal parameter, or a `struct` field; D_s is only suitable for the declaration of a type synonym. Statements are represented by S, there exist three of them: an expression-statement, consisting of an expression E followed by a semi-colon; a declaration-statement, where a semi-colon follows a declaration D; and a return-statement. This latter comes accompanied with an expression E, since, in μC, a function is required to have a valued return - given that we do not offer any control-flow statements, the last statement of a function body will always be `return` E;. This requirement imposes no loss of generality, but it makes our formalism simpler, since we avoid dealing with an ignored return, as it happens in C with a `void` function.

Among expressions, which are denoted by E, we subsume all binary ones by means of an $\oplus$. This symbol, in μC, denotes four different operators: assignment,

[5]μC lives in the same repository of PsycheC: `https://github.com/ltcmelo/psychec`.

=; addition, +; division, /; and logical OR, ||. As we discuss in Section 3.4, those are good representatives of the binary operations present in C. The pointer-related syntaxes are: dereferencing (also known as indirection), field access, and retrieval of an object's address. A *program identifier* is designated by x. Literals, which are `int` and `double`, are represented by ℓ. A type is denoted by τ. Every type can be *modified* to a pointer $\tau *$, *qualified* through `const`, or *composed* into a function or record, like in `struct` x $\{D^\star\}$. A *named type* $\mathcal{T}_n$ designates a type constructed from a name n. The name of a type is a string consisting of a program identifier, possibly prefixed with the keyword "struct"[6]. Not every type has a name, only records and those resulting from a `typedef` declaration. As usual, α is a type variable.

$$\begin{array}{lcll}
P & ::= & D_s^\star\ F^\star & ;\text{Program} \\
F & ::= & \tau\ f\ \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} & ;\text{Function} \\
D_s & ::= & \texttt{typedef}\ \tau\ \tau'\texttt{;} & ;\text{Type synonym} \\
D & ::= & \tau\ x & ;\text{Var. or param.} \\
E & ::= & \ell & ;\text{Literal} \\
 & | & x & ;\text{Identifier-expr.} \\
 & | & E\texttt{->}x & ;\text{Field access} \\
 & | & *E & ;\text{Dereference} \\
 & | & \&E & ;\text{Address-of} \\
 & | & E \oplus E & ;\text{Bin. op.: } \texttt{=}, \texttt{+}, \texttt{/}, \texttt{||}
\end{array}
\qquad
\begin{array}{lcll}
S & ::= & D; & ;\text{Decl. statement} \\
 & | & E; & ;\text{Expr. statement} \\
 & | & \texttt{return}\ E; & ;\text{Return} \\
\tau & ::= & \texttt{int} & ;\text{Integer} \\
 & | & \texttt{double} & ;\text{Floating point} \\
 & | & \tau * & ;\text{Pointer type} \\
 & | & \texttt{const}\ \tau & ;\text{Qualified type} \\
 & | & \tau \rightarrow \tau & ;\text{Function type} \\
 & | & \texttt{struct}\ x\ \texttt{\{}D^\star\texttt{\}} & ;\text{Record type} \\
 & | & \mathcal{T}_n & ;\text{Named type} \\
 & | & \alpha & ;\text{Type variable}
\end{array}$$

Figure 3.1. The syntax of μC. Toward cleanliness of the presentation, we omit the comma separating formal parameters and the semi-colon between fields of a record; x and f range over program identifiers. The μC language has additional rules that are not reflected by this grammar. For instance, the τ in a local declaration may not be an α (this type is reserved for internal use within constraints - every type specifier has an associated type variable), the τ' in a `typedef` must always be in the form $\mathcal{T}_n$ (*i.e.*, the type being defined must be given a name), and the last statement in a function body must be a `return` E`;`. Other restrictions are discussed in the text when appropriate. The implementation corresponding to this figure is available in Appendix A.1.

[6]In this context, we emphasize a non-obvious, and often misunderstood, difference between C++ and C. With the former language, the declaration `struct T {...}` defines a type *named* "T". But in C, and as we adhere to in μC, the name defined by that same declaration is actually "struct T". An effective idiom used by C programmers that puts both the name "T" and "struct T" into scope is to define such aggregate type combined with a `typedef`, as in `typedef struct T {...} T;`.

3.1.1 Programming Style

A μC program looks quite similar to a C program. Therefore, we do not further elaborate an specification for our language. Yet, the type system of μC has a quite distinguishing feature: in our language, it is **optional** to provide definitions for types (*e.g.* that of a `struct` or the declaration of a `typedef` synonym) used throughout the program. For instance, $\mathcal{P}_r$ = `T1 f() {T2 x; x->y=42; return x->y;}`, despite missing all the information about `T1` and `T2`, is a valid program. Such flexibility is permitted in μC because its type system is equipped with type inference.

Given that our language supports type inference, one may wonder why μC expects that variables are declared, and why its syntax requires that such declarations are accompanied by a type annotation. The reason for this design is to have a subset of C. After all, our main proposition is to provide a type inference technique for the latter language, thus we cannot expect that the syntax of existing C programs is modified. Apart from demanding the declaration of program variables[7] and that such declarations are accompanied by type annotations, writing programs that lack definitions for types approximates μC to the niche of languages like Javascript, Python, and SELF [Ungar and Smith, 1987].

In typical dynamic languages, the definition of a type may happen implicitly[8], based on how an object of such a type is used throughout the program. This same programming style can be employed in our language. Reconsidering program $\mathcal{P}_r$ = `T1 f() {T2 x; x->y=42; return x->y;}`, mentioned earlier, our type inference creates the definition of `T2` as `typedef struct T2 {int y;} T2;`, and makes `T1` a synonym of `int` as in `typedef int T1;`. Once those declarations are available, a μC program can be compiled just like C program would be. We refer to a program like $\mathcal{P}_r$, where type declarations are absent, as a *reduced* program; programs that originally contain all declarations necessary for compilation are referred to as *self-contained* ones.

Definition 4 (Reduced and Self-contained Programs)**.** Let $\mathcal{P}$ be a valid μC program as according to the syntax of Figure 3.1. If and only if there is at least one direct or transitive (through the application of type constructors) use, in $\mathcal{P}$, of a named type whose definition is not available, then we say that $\mathcal{P}$ is a *reduced* μC program. Otherwise, we say that $\mathcal{P}$ is a *self-contained* μC program. $\diamond$

[7]In PsycheC, it is not mandatory that the declaration of a program variable exists, in order to account for global variables.

[8]It is not relevant to this discussion whether the underlying type construction mechanism is class- or prototype-based.

Example 3. Program $\mathcal{P}_1$ = `T1 f() {T1* x; x=0; return *x;}` is a reduced program, but $\mathcal{P}_2$ = `typedef int T2; T2 f() {T2* x; x=0; return *x;}` is not. $\mathcal{P}_2$ is a self-contained program.

3.2 The Syntax of μC Constraints

The formulation of our type inference closely follows that of Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005]. Parts where a divergence occurs are explicitly pointed. In this Section, we present the syntax and semantics of constraints; afterwards, in Section 3.4, the constraint generation rules are discussed; and later, in Section 4.2, our constraint solving process is explained - the practice of establishing program properties through a constraint system is employed by Nielson *et al.* [Nielson et al., 2005, Ch.3-5] as well, but in the context of static analysis.

A *constraint language* has two groups of syntactic elements: one to designate types, whose constructs we borrow from the definition of μC, and another for denoting constraints themselves. The latter is presented in Figure 3.2 and implemented in Appendix A.2. We use K for a constraint; $\top$ and $\bot$ stand for *truth* and *falsity*, respectively. $K_1 \wedge K_2$ is the conjunction of two constraints; existential quantification is denoted by $\exists\alpha.K$, which means that a type variable α exists in constraint K; $\textit{def } x : \alpha \textit{ in } K$ is the introduction of an identifier that represents a program variable, by means of an *explicit substitution*; the instantiation of a type is given by $\textit{typeof}(x, \tau)$; constraint $\textit{syn } \tau \textit{ as } \alpha$ is necessary for us to match the names of a type used in a declaration to that of the eventually instantiated type; $\textit{has}(\alpha, x : \tau)$ is the field membership relation, where we refer to α as the *enclosing* type, and to τ as the *field type.*

Besides $\textit{def } x : \alpha \textit{ in } K$, our syntax provides another constraint that binds a type to an identifier: $\textit{fun } f : \tau \rightarrow \alpha \textit{ in } K$. The principle behind this latter constraint is the same as that of the former. But, there is a technical, although subtle, difference between the two: the type bound to *fun* is always of an arrow form; in *def*, that is never the case. Because formal parameters are inherently captured by a *def* constraint, the remaining component of a function that must be addressed is its return type. Thus, offering an specific *fun* constraint brings certain benefits both from a formalism and implementation standpoint. Those should become clear once we present the semantics of constraints, in Section 3.3, our generators, in Section 3.4.

It remains to present our constraints for a *type predicate*, *i.e.* the way we compare one type to another. At this point, we expand the work of Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005], in that our language ac-

$$
\begin{array}{rcll}
K & ::= & \top & \text{; Truth} \\
 & \mid & \bot & \text{; Falsity} \\
 & \mid & K_1 \wedge K_2 & \text{; Conjunction} \\
 & \mid & \exists \alpha . K & \text{; Existential quantification} \\
 & \mid & \mathit{def}\ x : \alpha\ \mathit{in}\ K & \text{; Variable introduction} \\
 & \mid & \mathit{fun}\ f : \tau \rightarrow \alpha\, \mathit{in}\ K & \text{; Function introduction} \\
 & \mid & \mathit{typeof}(x, \alpha) & \text{; Type instantiation} \\
 & \mid & \mathit{syn}\ \tau\ \mathit{as}\ \alpha & \text{; Type synonym} \\
 & \mid & \mathit{has}(\alpha, x : \tau) & \text{; Field membership} \\
 & \mid & \tau_1 \equiv \tau_2 & \text{; Type equivalence} \\
 & \mid & \tau_1 \leq \tau_2 & \text{; Type inequality}
\end{array}
$$

Figure 3.2. The syntax of constraints for μC. We slightly abuse notation by restricting certain types to a specific form, *e.g.* that τ is a type variable, α, or an arrow, $\tau \rightarrow \alpha$. This choice should facilitate the reader's comprehension and can be justified by our constraint generation rules, yet to be presented in Section 3.4. Appendix A.2 contains the implementation of the constraints in this figure.

counts for relations both in the form of equivalence, $\equiv$, and of inequality, $\leq$. The reason for such expansion is to account for implicit conversions that exist in C. For instance, an `int` can be assigned to a `double`, but the reverse can potentially lead to value truncation[9]. Furthermore, we model the behavior between pointer types as a subtyping relation, by interpreting $\tau*$ as a subtype of `const` $\tau*$. The rationale under such approach is explained in in Section 3.3.

Given the presence of an inequality predicate (which ultimately implies subtyping) in our constraint language, a question that may arise is whether we are able to keep our solving method within the bounds of an *equality-only model*, as in the framework of Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005]. The answer to this question is yes. Our constraint solver operates on subtyping relations as if those were equivalences. However, for the processing of inequalities we employ an *ordering* criterion that ensures that a subtype will be (i) lifted to is base type if necessary, or (ii) preserved as such, whenever possible. The details about this process are show in Section 4.2.

[9]Even though such behavior is specified in the C standard[ISO-Standard, 2011]{§6.3.1.4-1}, both gcc and clang are permissive in this regard, only producing a warning in such a case.

3.3 The Semantics of μC Constraints

Reasoning in terms of constraints requires that we give meaning to them. Toward this task, it is necessary to describe how the syntax of Figure 3.2 corresponds to types of μC; not to arbitrary types but to *ground types*, those that are free from type variables, *i.e.*, $\mathit{ftv}(\tau) = \varnothing$. Ground types are instantiated by means of a *substitution*, which is denoted by $[\alpha \mapsto \tau]$. The exact procedure under which such instantiation happens is called an *application*. Such application may be over a constraint, for which notation $[\alpha \mapsto \tau]\ K$ is employed, or over a type, as in $[\alpha \mapsto \tau]\tau'$.

The exact representation of a substitution may vary. For instance, our implementation in μC is different than the one in PsycheC. While in the latter we use a map that implicitly carries relations among types, in the former, we have a pair composed by a *stamp* and a type. Encoding type variables through a stamp, which is just a natural number, is a convenience that facilitates the formalism. We adopt this approach by following both Dubois and Ménissier-Morain [Dubois and Menissier-Morain, 1999] and Naraschewski and Nipkow [Naraschewski and Nipkow, 1999]. Nevertheless, toward clarity of the presentation, we abstract away the notion of a stamp. μC's implementation of a substitution, alongside with its related operations, appear in Appendix A.3.

The C language adopts a nominal type system, thus the equivalence among types cannot be established on the basis of their structure. It is necessary, therefore, the ability to identify types by their name. When the syntax of μC was introduced in Section 3.1, we mentioned that builtins, records, and `typedef`ed types are considered named types. However, a type like `int*` does not have a name. Yet, the character string "int*" could still uniquely identify it. We refer to such encoding of a type as a *typeid*. Definition 5 formalizes this concept.

Definition 5 (Type Identification)**.** Let τ be a type. We refer to the character string that uniquely identifies it as the *typeid* of τ. To obtain the typeid of a type, function $\widehat{\ }$ (hat), defined in Figure 3.3, is applied to it. Such application is denoted by $\widehat{\tau}$. ◇

At this point, we are ready to present the semantics of our constraints. These semantics, outlined in Figure 3.4 and implemented in Appendix A.6, are a collection of requirements that must be satisfied by a type instantiation algorithm (in this work, a constraint solver) that is capable of producing well typed programs. Our rules shall be interpreted as a judgement that reads: *K is satisfiable if there exist ϕ, ψ, and Θ such that $\phi, \psi, \Theta \models K$ holds*, where $\models$ is the *satisfaction predicate*, and ϕ, ψ, and Θ are map data structures. Definition 6 formalizes this statement.

$$
\begin{aligned}
&\widehat{\texttt{int}} && = \text{“int”} \\
&\widehat{\texttt{double}} && = \text{“double”} \\
&\widehat{\tau *} && = \widehat{\tau} +\!+ \text{“}*\text{''} \\
&\widehat{\texttt{const}\,\tau} && = \text{“const ''} +\!+ \widehat{\tau} \\
&\widehat{\tau_1 \to \tau_2} && = \widehat{\tau_1} +\!+ \ \text{“}(*)\text{''} +\!+ \widehat{\tau_2} \\
&\widehat{\texttt{struct}\ \mathrm{T}\ \texttt{\{}D^\star\texttt{\}}} && = \text{“struct T''} \\
&\widehat{\mathcal{T}_n} && = \text{“n”} \\
&\widehat{\alpha_i} && = \text{“i”}
\end{aligned}
$$

Figure 3.3. The $\widehat{}$ (hat) function. The application of this function over a type τ, denoted by $\widehat{\tau}$, yields a character string that uniquely identifies such a type. This string is referred to the *typeid* of τ. In this figure, ++ is the list concatenation operator. The implementation of $\widehat{}$ appears in Appendix A.4.

Table 3.1. The description of the ϕ, ψ, and Θ map data structures.

Map	From	To	Element
ϕ	type variable	type	$\{\alpha, \tau\}$
ψ	program identifier	type	$\{x, \tau\}$
Θ	*typeid*	type (definition)	$\{\texttt{typeid}, \tau\}$

Definition 6 (Satisfaction Judgment)**.** Let K be a constraint as according to the grammar of Figure 3.2. We say that K is *satisfiable* if and only if the judgment $\phi, \psi, \Theta \models K$ holds, as required by the semantics of Figure 3.4, where ϕ, ψ, and Θ are the data structures in Table 3.1.$\diamond$

Prior to discussing the semantics of our constraints, let us establish further notation. Given a map M, which may be any of ϕ, ψ, and Θ, whose key is k, we denote: (i) the *introduction* of a new key-value pair to M by $M \cup \{k,\ \tau\}$; (ii) the *lookup* of the type τ associated with an existing key in M by $M(k)$; (iii) and the *assignment* of a key from M by $M[k \mapsto \tau]$ – the similarity of this latter notation to that of a substitution is intended.

We now go over each of the rules in Figure 3.4. *KTrue* is a tautology: it holds, regardless of ϕ, ψ, or Θ. *KAnd* requires that both K_1 and K_2 hold, independently. With *KEx*, we guarantee that every type variable introduced by an existential quantifier is bound to a ground type, regardless of which exact type that is. However, due to the possible absence of declarations, orphan type variables may remain uninstantiated after the solving process. Nevertheless, as further explained in Section 4.2.6, it is always

$$\frac{}{\phi, \psi, \Theta \models \top}\ \{KTrue\} \qquad \frac{\phi, \psi, \Theta \models K_1 \quad \phi, \psi, \Theta \models K_2}{\phi, \psi, \Theta \models K_1 \wedge K_2}\ \{KAnd\}$$

$$\frac{\phi[\alpha \mapsto \tau], \psi, \Theta \models K \quad \mathit{ftv}(\tau) = \varnothing}{\phi, \psi, \Theta \models \exists \alpha . K}\ \{KEx\} \qquad \frac{\phi, \psi[x \mapsto \phi(\alpha)], \Theta \models K}{\phi, \psi, \Theta \models \mathit{def}\ x : \alpha\ \mathit{in}\ K}\ \{KDef\}$$

$$\frac{\phi, \psi[f \mapsto (\tau \rightarrow \phi(\alpha))], \Theta \models K}{\phi, \psi, \Theta \models \mathit{fun}\ f : \tau\ \rightarrow \alpha\ \mathit{in}\ K}\ \{KFun\} \qquad \frac{\phi, \psi, \Theta \models \psi(x) \equiv \alpha}{\phi, \psi, \Theta \models \mathit{typeof}(x, \alpha)}\ \{KInst\}$$

$$\frac{\texttt{field}(x, \Theta(\widehat{\phi(\alpha)})) = \tau' \quad \phi, \psi, \Theta \models \tau' \equiv \tau}{\phi, \psi, \Theta \models \mathit{has}(\alpha, x : \tau)}\ \{KHas\} \qquad \frac{\phi, \psi, \Theta[\widehat{\tau} \mapsto \tau'] \models \tau' \equiv \alpha}{\phi, \psi, \Theta \models \mathit{syn}\ \tau\ \mathit{as}\ \alpha}\ \{KSyn\}$$

$$\frac{\phi \vdash \tau_1 <: \tau_2}{\phi, \psi, \Theta \models \tau_1 \equiv \tau_2}\ \{KEq\} \qquad \frac{\phi \vdash \tau_1 <: \tau_2}{\phi, \psi, \Theta \models \tau_1 \leq \tau_2}\ \{KIq\}$$

Figure 3.4. The semantics of constraints. ϕ maps type variables to types, ψ maps program identifiers to types, and Θ maps a *typeid* (see Definition 5) to the definition of a type. Notation $\widehat{\tau}$ is used to refer to the typeid of τ. ϕ, ψ, and Θ are maps as according to Definition 6. Function `field` retrieves, from a record, the type of the field whose name is passed as the first argument. Appendix A.6 shows the implementation of our semantics validation.

possible to bind such orphans in a safe manner.

KDef introduces a variable, just like *KFun* introduces a function. Both rules are quite similar in that they bind a program identifier to a type. Yet, because each formal parameter is already subject to *KDef*, our main interest on *KFun* concerns a function's return type. This fact, which is the reason for why we separate the two, also allows us provide clearer formalism and a simpler implementation (it is always safe to pattern match a *fun* constraint against an arrow type, and whose return is a type variable). *KInst* is a type instantiation rule as a well. In particular, it relies on the fact that any program variable x has been previously introduced by *KDef.*

KSyn has interesting origins. Typically, in a language with native type inference, program variables may be declared without an explicit type annotation. This is not the case in μC, where type annotations are always present, it is just the definition of such annotated types that may be missing. Therefore, our type inference must not only reconstruct a type such that it makes the program well typed, but the typeid of the inferred type must match that used in a given declaration. As an analogy, *KSyn* is in our constraint language what `typedef` is in C.

A function named `field` is used by *KHas.* This function is only defined for records, what implies that $\phi(\alpha)$ must be a `struct` type whose definition is available in Θ through typeid in question; it is an undefined behaviour to invoke this function with any other type as an argument, The declaration of `field` takes, in addition to a record, an identifier x. The result of an invocation to this function is the type, τ', of field x, in the record given as argument. Because every generated $has(\alpha, x : \tau)$ constraint captures the field name within it, a declaration of such `struct` is expected, and τ must be equivalent to τ'.

The last two remaining rules, *KEq*, and *KIq*, have all one thing in common: their premises rely on the type predicate $\tau_1 <: \tau_2$, which is defined under a syntactical judgement of ϕ, exclusively. One should not confuse the $\tau_1 \leq \tau_2$ constraint with the $\tau_1 <: \tau_2$ type predicate; the former, as well as $\tau_1 \equiv \tau_2$, establishes a **requirement** that is to be **satisfied** by the latter. But where does subtyping comes from in μC or C? The answer to this question is twofold: first, as it can be show in Figure 3.5, our semantics allows a conversion from `int` to `double`, but not the other way around - this relation is a form of *atomic* subtyping [Mitchell, 1991]; second, subtyping relations also emerge from the way we model pointer relations.

We interpret a pointer type, $\tau*$, as a subtype of a `const` pointer type, `const` $\tau*$. This modeling establishes a partial order among types. An effect of such ordering is that a $\tau*$ can be directly assigned to a `const` $\tau*$, but an assignment on the opposite direction demands an explicit cast. Even though it is counter-intuitive to think of non-`const` pointers being a subset of `const` pointers, this subtyping relation, which can also be found in the work of Foster *et al.* [Foster et al., 1999], meets Liskov's substitution principle: a $\tau*$ *can safely be used in a context where a* `const` $\tau*$ is expected.

The type predicate rules of our constraints are shown in Figure 3.5. Their implementation appear in Appendix A.7. Rules *SInt*, *SDbl*, *SCon*, *SSpc*, *SVr*, and *SVl* are straightforward – *SVrv* and *SVlv* are similar to the latter two, although they apply when $\phi(\alpha)$ is the identity mapping (as discussed in Section 3.5, the typing rules of μC ultimately require $ftv(\tau) = \varnothing$ for typings to be valid). But a few of the other rules might not be obvious at first sight. Let us being with *SCnv*, which is consequence of the fact that μC permits an implicit conversion from `int` to `double`. Note that a rule allowing such conversion in the other direction (*i.e.*, from `double` to `int`) does not exists, since that could lead to value truncation. Back in Figure 1.4, we have seen that a non-`const` value can be assigned to a `const` one, and that assigning to a `const` value is forbidden (`const` objects can only have their value specified during variable *initialization*). *SCl* reflects the former statement, while the latter fact is ensured by the absence of a predicate with $\phi \vdash \tau_1 <:$ `const` τ_2 in its conclusion.

About half of the type predicates in Figure 3.5 have their premises and/or conclusions in the form $\phi \vdash \lceil ... \rceil$. We employ this notation to express that the relation in question is *lifted* into a pointer type. The exact rules that triggers this lifting is *PPtr*: once it is observed that a given relation involves pointers, then slight different demands are imposed on the constraints that participate in such relation. Due to its importance, we highlight *PPtr*. However, the lifted versions of *PInt*, *PDbl*, *PSpc*, and *PCon*, look the same as their counterparts without $\lceil$ and $\rceil$, *SInt*, *SDbl*, *SSpc*, and *SCon*. That happens because these predicates are not affected by the presence of pointer types. But note that the semantics of lifted relations do not include a rule corresponding to *SCnv*. As opposed to `int` <: `double`, the relation int$*$ <: double$*$ is not valid - that would be an unsafe operation, since the memory cells dereferenced by those pointers are of different size. On the other hand, as shown by predicate *PCr*, we do allow a conversion from a non-`const` pointer type to a `const` pointer. Subtyping is further discussed in Section 4.

$$\frac{}{\phi \vdash \texttt{int} <: \texttt{int}}\{SInt\} \qquad \frac{}{\phi \vdash \lceil \texttt{int} <: \texttt{int} \rceil}\{PInt\} \qquad \frac{}{\phi \vdash \texttt{int} <: \texttt{double}}\{SCnv\}$$

$$\frac{\phi \vdash \tau_1 <: \tau_2}{\phi \vdash \texttt{const}\ \tau_1 <: \texttt{const}\ \tau_2}\{SCon\} \qquad \frac{\phi \vdash \tau_1 <: \tau_2}{\phi \vdash \texttt{const}\ \tau_1 <: \tau_2}\{SCl\} \qquad \frac{\mathcal{T}_{n_1} = \mathcal{T}_{n_2}}{\phi \vdash \mathcal{T}_{n_1} <: \mathcal{T}_{n_2}}\{SSpc\}$$

$$\frac{}{\phi \vdash \texttt{double} <: \texttt{double}}\{SDbl\} \qquad \frac{}{\phi \vdash \lceil \texttt{double} <: \texttt{double} \rceil}\{PDbl\} \qquad \frac{\mathcal{T}_{n_1} = \mathcal{T}_{n_2}}{\phi \vdash \lceil \mathcal{T}_{n_1} <: \mathcal{T}_{n_2} \rceil}\{PSpc\}$$

$$\frac{\phi \vdash \lceil \tau_1 <: \tau_2 \rceil}{\phi \vdash \lceil \texttt{const}\ \tau_1 <: \texttt{const}\ \tau_2 \rceil}\{PCon\} \qquad \frac{\phi \vdash \lceil \tau_1 <: \tau_2 \rceil}{\phi \vdash \lceil \tau_1 <: \texttt{const}\ \tau_2 \rceil}\{PCr\} \qquad \frac{\phi \vdash \lceil \tau_1 <: \tau_2 \rceil}{\phi \vdash \tau_1 * <: \tau_2 *}\{PPtr\}$$

$$\frac{\phi \vdash \tau <: \tau' \quad \tau' = \phi(\alpha) \quad ftv(\tau') = \varnothing}{\phi \vdash \tau <: \alpha}\{SVr\} \qquad \frac{\phi \vdash \lceil \tau <: \tau' \rceil \quad \tau' = \phi(\alpha) \quad ftv(\tau') = \varnothing}{\phi \vdash \lceil \tau <: \alpha \rceil}\{PVr\}$$

$$\frac{\phi \vdash \tau' <: \tau \quad \tau' = \phi(\alpha) \quad ftv(\tau') = \varnothing}{\phi \vdash \alpha <: \tau}\{SVl\} \qquad \frac{\phi \vdash \lceil \tau' <: \tau \rceil \quad \tau' = \phi(\alpha) \quad ftv(\tau') = \varnothing}{\phi \vdash \lceil \alpha <: \tau \rceil}\{PVl\}$$

$$\frac{\phi(\alpha) = \alpha}{\phi \vdash \tau <: \alpha}\{SVrv\} \qquad \frac{\phi(\alpha) = \alpha}{\phi \vdash \alpha <: \tau}\{SVlv\} \qquad \frac{\phi(\alpha) = \alpha}{\phi \vdash \lceil \tau <: \alpha \rceil}\{PVrv\} \qquad \frac{\phi(\alpha) = \alpha}{\phi \vdash \lceil \alpha <: \tau \rceil}\{PVlv\}$$

Figure 3.5. The type predicate rules of our constraints semantics. This figure is divided into two groups: relations that involve non-pointer types, and relations where the involved types are pointers. The latter are those surrounded by $\lceil ... \rceil$. We refer to them as *lifted* relations. The highlighted (and to the right) rule, *PPtr*, is responsible for lifting the predicates whenever it can be observed that the participating types are pointers. Our semantics accepts a subtyping relation where one of the types is the identity mapping – although, as discussed in Section 3.5, the typing rules of μC ultimately require $ftv(\tau) = \varnothing$ for typings to be valid. The implementation of these rules is in Appendix A.7.

3.4 Syntax Directed Constraint Generation

The constraints of a μC program are generated by traversing its *abstract syntax tree* (AST). To produce such AST, we put into practice the disambiguation algorithm presented in Section 2. During the traversal, constraints are generated for every construct of μC (*i.e.*, those in Figure 3.1), as according to the *generators* in Figure 3.6. There are four main categories of them: (i) a top-level generator, $\langle\langle\, P, \mathcal{M}\,\rangle\rangle_p$, that will handle `typedef` declarations and, subsequently, invoke a generator for each of the program's function, ensuring that the return type is propagated; (ii) a generator for functions, $\langle\langle\,$ `(`$D^\star$`)` `{`$S^\star$`}` $: \tau_r,\, \mathcal{M}\,\rangle\rangle_f$, that consumes their formal parameters and, eventually, delegates to a generator that will deal with their bodies; (iii) a generator for statements, $\langle\langle\, S^\star : \tau_r, \mathcal{M}\,\rangle\rangle_s$, which, besides handling each statement, asserts that a function's return type is matched by a `return` E`;` and (iv) an expression generator, $\langle\langle\, E : \tau, \mathcal{M}\,\rangle\rangle_e$. The implementation of our constraint generation is available in Appendix A.8.

In addition to the main generators, we have a few auxiliary ones. The first thereof to be explained is $\langle\langle\, \tau, \alpha\,\rangle\rangle_{bs}$. The role of this generator is to ***build*** a ***synonym*** for every type that is used in a declaration or as the return of a function, even when such a type is indirectly mentioned through a pointer modifier or a `const` qualifier, respectively, as in $\tau *\ x$; and in `const` $\tau\ x$;. Such synonyms are necessary because the type system of C is a nominal one, so we need a correlation between type variables and the actual name of types in the program. The implementation of this generator is available in Appendix A.9 and illustrated in Figure 3.7. The other two auxiliary generators are more involving. Prior to describing their behavior, we need to explain our lattice of shapes, which is related to the $\mathcal{M}$ that appears in the generators from Figure 3.6.

3.4.1 A Lattice of Shapes

A binary expression, $E_1 \oplus E_2$, from μC poses a challenge to our constraint generation procedure: except for division, the sides of operations $+$, $||$, and $=$ may involve types that are not necessarily unifiable. For instance, in a logical OR expression, E_1 can be of type `int` while E_2 can be a pointer like `int`$*$. As another example, recall that `0` is the *null pointer constant* [ISO-Standard, 2011]{§6.3.2.3.3} in the C language. Hence, a value of type `int` can be assigned to arithmetic and pointer types as well. Such an assignment may also implicitly happen between the type variables we create for the return of a function and that of a return-statement, *e.g.* as in $\mathcal{P}$ = `int`$*$ `f() { return 0; }`. Therefore, our generators must be able to distinguish between values of pointer and arithmetic types, avoiding a (wrong) unification between them.

$\langle\!\langle\, P,\, \mathcal{M} \,\rangle\!\rangle_p$

$$
\begin{aligned}
\langle\!\langle\, \varnothing,\, \mathcal{M} \,\rangle\!\rangle_p &= \top \\
\langle\!\langle\, \texttt{typedef}\ \tau\ \tau';\ D_s^\star\ F^\star,\, \mathcal{M} \,\rangle\!\rangle_p &= \exists\alpha.\, \mathit{syn}\ \tau'\ \mathit{as}\ \alpha \wedge \tau \equiv \alpha \wedge \langle\!\langle\, D_s^\star\ F^\star,\, \mathcal{M} \,\rangle\!\rangle_p \\
\langle\!\langle\, \tau_r\ f(D^\star)\ \texttt{\{}S^\star\texttt{\}}\ F^\star,\, \mathcal{M} \,\rangle\!\rangle_p &= \exists\alpha.\, \langle\!\langle\, \tau_r,\, \alpha \,\rangle\!\rangle_{bs} \wedge \mathit{fun}\, f : (\tau_1 \to \ldots \to \tau_n) \to \alpha\, \mathit{in} \\
&\qquad \langle\!\langle\, \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}}, \alpha,\, \mathcal{M} \,\rangle\!\rangle_f \wedge \langle\!\langle\, F^\star,\, \mathcal{M} \,\rangle\!\rangle_p
\end{aligned}
$$

$\langle\!\langle\, \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_f$

$$
\begin{aligned}
\langle\!\langle\, \texttt{(}\tau\ x, D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_f &= \exists\alpha.\, \langle\!\langle\, \tau,\, \alpha \,\rangle\!\rangle_{bs} \wedge \mathit{def}\, x : \alpha\ \mathit{in}\ \langle\!\langle\, \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_f \\
\langle\!\langle\, \texttt{()}\ \texttt{\{}S^\star\texttt{\}} : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_f &= \langle\!\langle\, S^\star : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s
\end{aligned}
$$

$\langle\!\langle\, S^\star : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s$

$$
\begin{aligned}
\langle\!\langle\, \tau\ x;\ S^\star : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s &= \exists\alpha.\, \langle\!\langle\, \tau,\, \alpha \,\rangle\!\rangle_{bs} \wedge \mathit{def}\, x : \alpha\ \mathit{in}\ \langle\!\langle\, S^\star : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s \\
\langle\!\langle\, E;\ S^\star : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s &= \exists\alpha.\, \langle\!\langle\, E : \alpha,\, \mathcal{M} \,\rangle\!\rangle_e \wedge \langle\!\langle\, S^\star : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s \\
\langle\!\langle\, \texttt{return}\ E;\, : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_s &= \exists\alpha.\, \langle\!\langle\, \mathcal{M}(\$\mathit{ret}), \tau_r, \mathcal{M}(E), \alpha, = \,\rangle\!\rangle_{kd} \wedge \langle\!\langle\, E : \alpha,\, \mathcal{M} \,\rangle\!\rangle_e
\end{aligned}
$$

$\langle\!\langle\, E : \tau,\, \mathcal{M} \,\rangle\!\rangle_e$

$$
\begin{aligned}
\langle\!\langle\, \ell : \tau,\, \mathcal{M} \,\rangle\!\rangle_e &= \rho(\ell) \equiv \tau \\
\langle\!\langle\, x : \tau,\, \mathcal{M} \,\rangle\!\rangle_e &= \mathit{typeof}(x, \tau) \\
\langle\!\langle\, E\texttt{->}x : \tau,\, \mathcal{M} \,\rangle\!\rangle_e &= \exists\alpha_1\alpha_2\alpha_3.\ \ \mathit{has}(\alpha_2, x : \alpha_3) \wedge \alpha_1 \equiv {\alpha_2}^* \wedge \alpha_3 \equiv \tau \wedge \langle\!\langle\, E : \alpha_1,\, \mathcal{M} \,\rangle\!\rangle_e \\
\langle\!\langle\, {*}\,E : \tau,\, \mathcal{M} \,\rangle\!\rangle_e &= \exists\alpha.\ \alpha \equiv \tau^* \wedge \langle\!\langle\, E : \alpha,\, \mathcal{M} \,\rangle\!\rangle_e \\
\langle\!\langle\, \&\,E : \tau,\, \mathcal{M} \,\rangle\!\rangle_e &= \exists\alpha_1\alpha_2.\ \alpha_1 \equiv {\alpha_2}^* \wedge \alpha_1 \equiv \tau \wedge \langle\!\langle\, E : \alpha_2,\, \mathcal{M} \,\rangle\!\rangle_e \\
\langle\!\langle\, E_1 \oplus E_2 : \tau,\, \mathcal{M} \,\rangle\!\rangle_e &= \exists\alpha_1\alpha_2.\, \langle\!\langle\, E_1 : \alpha_1,\, \mathcal{M} \,\rangle\!\rangle_e \wedge \langle\!\langle\, E_2 : \alpha_2,\, \mathcal{M} \,\rangle\!\rangle_e \wedge \\
&\qquad \langle\!\langle\, \mathcal{M}(E_1), \alpha_1, \mathcal{M}(E_2), \alpha_2, \oplus, \tau \,\rangle\!\rangle_{sel} \wedge \\
&\qquad \langle\!\langle\, \mathcal{M}(E_1), \alpha_1, \mathcal{M}(E_2), \alpha_2, \oplus \,\rangle\!\rangle_{kd}
\end{aligned}
$$

Figure 3.6. The constraint *generators* for μC. There are four principal ones: a whole-program generator, $\langle\!\langle\, P, \mathcal{M} \,\rangle\!\rangle_p$; a generator for functions, $\langle\!\langle\, \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r,\, \mathcal{M} \,\rangle\!\rangle_f$; another for statements, $\langle\!\langle\, S^\star : \tau_r, \mathcal{M} \,\rangle\!\rangle_s$; and an expression generator, $\langle\!\langle\, E : \tau, \mathcal{M} \,\rangle\!\rangle_e$. In this figure, we abbreviate the types in the formal parameters $D^\star$ of a function by $\tau_1 \to \ldots \to \tau_n$. Therefore, the *fun* constraint of a function that returns τ_r is written as $\mathit{fun}\, f : (\tau_1 \to \ldots \to \tau_n) \to \tau_r\, \mathit{in}\, K$. We use an artificial expression $\$\mathit{ret}$ to represent, in $\mathcal{M}$, the return type of a function. These generators are implemented in Appendix A.8.

A possible strategy to deal with this difficulty would be to have our generators entirely discard constraints of binary expressions. However, such naïve approach would incur on the penalty that legitimate type relations get thrown away and, consequently, information that is potentially necessary to the type inference is lost. A smarter solution is needed. One might wonder why not simply looking up the types of an expression's operands in the symbol table. Indeed, that will work for self-contained

$$\begin{aligned}
\langle\langle\, \texttt{const}\,\tau,\, \alpha\,\rangle\rangle_{bs} &= \exists\alpha'.\, syn\ \texttt{const}\,\tau\ as\ \alpha \wedge syn\ \tau\ as\ \alpha' \wedge \texttt{const}\,\alpha' \equiv \alpha \wedge \langle\langle\, \tau,\, \alpha'\,\rangle\rangle_{bs} \\
\langle\langle\, \tau*,\, \alpha\,\rangle\rangle_{bs} &= \exists\alpha'.\, syn\ \tau*\ as\ \alpha \wedge syn\ \tau\ as\ \alpha' \wedge \alpha'* \equiv \alpha \wedge \langle\langle\, \tau,\, \alpha'\,\rangle\rangle_{bs} \\
\langle\langle\, \tau,\, \alpha\,\rangle\rangle_{bs} &= syn\ \tau\, as\ \alpha
\end{aligned}$$

Figure 3.7. Auxiliary generator that *builds synonyms* for types that appear in variable/parameter declarations and in the return of functions. Equivalence constraints for the deconstructed types are produced as well. The implementation of $\langle\langle\, \tau, \alpha\,\rangle\rangle_{bs}$ is available in Appendix A.9.

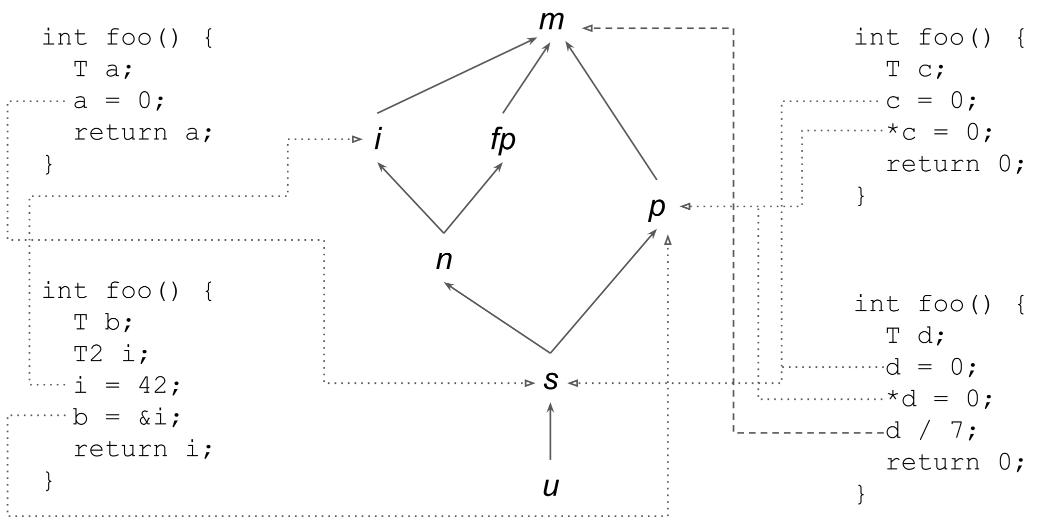
μC programs (see Definition 4). Yet, relying on semantic information is, in general, not an option, because reduced μC programs may lack the declaration of such types.

To deal with the absence of semantic information during constraint generation, we classify all expressions[10] in a program according to a lattice of *shapes*, $\mathcal{L} = \langle\{u, s, p, n, i, fp, m\}, <, \vee, \bot = u, \top = m\rangle$, where *u=undefined*, *s=scalar*, *p=pointer*, *n=numeric*, *i=integral*, *fp=floating-point*, and *m=malformed* (non-scalar types have shape u). A shape of our lattice, referred to as S, resembles a shape as proposed by Pottier and Régis-Gianas [Pottier and Régis-Gianas, 2006], *i.e.*, a type with holes (unspecified parts). But, in the latter case, a shape is the result of an inference algorithm; while in $\mathcal{L}$, shape information is an input to our type inference. Every C type has a shape[11], but not all shapes are C types, *e.g.*, an expression can be classified as p even though the underlying type of the pointer is unknown. Figure 3.8 gives our lattice's partial order and illustrates how program expressions are mapped to it.

The criteria employed in our shape classification mimics restrictions that the C language imposes on operands of expressions. Figure 3.9 shows, in a syntax-directed style, these exact rules. Their implementation appears in Appendix A.10. All rules carry along a table $\mathcal{M}$, which, after the traversal $[\![E, \mathsf{S}, \mathcal{M}]\!]$ of each syntax, is updated with a mapping from expression E to shape S. The shape associated with a given expression is retrieved by $\mathcal{M}(E)$. Because we need to correlate the shape $\mathcal{M}(E)$ of a return-statement `return` E`;`, with the return type, τ_r, of a function, an artificial expression $\$ret$ is created for the latter. Even though μC does not support all the operations available in C, our language is, from a type-wise standpoint, rich enough to allow the demonstration of our lattice's role. Of particular relevance is the selection of

[10]Variable declarations are as well considered for shape classification. For instance, through the declaration `int a;`, we know that, if an expression `a` if found, then it is of shape i. More interestingly, from a declaration like `T* p;`, we know that an expression `p` is of pointer type, regardless of the meaning of `T`.

[11]In PsycheC, our classification into shapes honors all the integral and floating-point types that exist in C, such as `short` and `long`, and `double` and `float`.



```
int foo() {
  T a;
  a = 0;
  return a;
}
```

```
int foo() {
  T b;
  T2 i;
  i = 42;
  b = &i;
  return i;
}
```

```
int foo() {
  T c;
  c = 0;
  *c = 0;
  return 0;
}
```

```
int foo() {
  T d;
  d = 0;
  *d = 0;
  d / 7;
  return 0;
}
```

Figure 3.8. The lattice of shapes, $\mathcal{L}$. The programs surrounding it illustrate the classification of expressions to shapes like s, i, p, and m. These mappings, consisting of an expression E and a shape S, are carried over by table $\mathcal{M}$ so that this information can be used during constraint generation.

binary expressions supported by μC:

- The division operation represents a category of binary expressions that require both operands to be of arithmetic type. The shape resulting out of E_1/E_2 is at least n.
- Expression $E_1 \,||\, E_2$ stands for operations that allow arithmetic and pointer types to be arbitrarily mixed. The return of logical OR, however, is of type `int`, thus of shape i.
- An addition, $E_1 + E_2$, subsumes binary expressions in which, not only arithmetic and pointer types can interact, but the type of one operand imposes a restriction on the type of the other: precisely, if E_1 is a p, then E_2 must be i (and vice-versa).
- An assignment, $=$, is a distinguished expression, making a category on its own.

The goal of the classification of expressions into shapes is that it performs a sort of "pre-inference". We can only proceed with constraint generation once table $\mathcal{M}$ has been populated. As seen in Figure 3.6, only the generators for a binary expression and a return-statement consult such table. Querying $\mathcal{M}$ for shape information is necessary for the workings of two auxiliary generators. One of them is $\langle\langle\, \mathsf{S}_1, \alpha_1, \mathsf{S}_2, \alpha_2,\, \mathcal{M}, \oplus \,\rangle\rangle_{kd}$, which we refer to as ***keep or drop***. Its purpose is to determine the nature of the relation

$$\llbracket \mathsf{0}, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M} \cup \{\ \mathsf{0} : s\} \qquad \llbracket \ell_i, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M} \cup \{\ell_i : i\} \qquad \llbracket \ell_{fp}, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M} \cup \{\ell_{fp} : fp\}$$

$$\llbracket x, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M} \cup \{x : \mathsf{S}\} \qquad \frac{\llbracket E\text{->}x, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'}{\llbracket E\text{->}x, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \cup \{E : p\}} \qquad \frac{\llbracket E, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'}{\llbracket \&E, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \cup \{\&E : p\}}$$

$$\frac{\llbracket E, \mathcal{S} \vee p, \mathsf{M} \rrbracket \rightarrow \mathcal{M}'}{\llbracket *E, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \cup \{*E : \mathsf{S}\}} \qquad \frac{\llbracket E_1, \mathsf{S} \vee n, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \qquad \llbracket E_2, \mathsf{S} \vee n, \mathcal{M}' \rrbracket \rightarrow \mathcal{M}''}{\llbracket E_1 \,/\, E_2, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'' \cup \{E_1 \,/\, E_2 : \mathcal{M}'(E_1) \vee \mathcal{M}''(E_2)\}}$$

$$\frac{\llbracket E_2, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \qquad \llbracket E_1, \mathsf{S} \vee \mathcal{M}'(E_2), \mathcal{M}' \rrbracket \rightarrow \mathcal{M}''}{\llbracket E_1 = E_2, L, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'' \cup \{E_1 = E_2 : \mathsf{S} \vee \mathcal{M}''(E_1)\}} \qquad \frac{\llbracket E_1, \mathsf{S} \vee s, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \qquad \llbracket E_2, \mathsf{S} \vee s, \mathcal{M}' \rrbracket \rightarrow \mathcal{M}''}{\llbracket E_1 \,||\, E_2, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'' \cup \{E_1 \,||\, E_2 : i\}}$$

$$\frac{\llbracket E_1, (\mathsf{S} \wedge s) \vee \{n, i, fp\}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \qquad \llbracket E_2, i, \mathcal{M}' \rrbracket \rightarrow \mathcal{M}'' \qquad \mathcal{M}'(E_1)\ \mathbf{is}\, p}{\llbracket E_1 + E_2, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'' \cup \{E_1 + E_2 : p\}}$$

$$\frac{\llbracket E_2, (\mathsf{S} \wedge s) \vee \{n, i, fp\}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \qquad \llbracket E_1, i, \mathcal{M}' \rrbracket \rightarrow \mathcal{M}'' \qquad \mathcal{M}'(E_2)\ \mathbf{is}\, p}{\llbracket E_1 + E_2, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}'' \cup \{E_1 + E_2 : p\}}$$

$$\frac{\llbracket E_1, (\mathsf{S} \wedge s) \vee \{n, i, fp\}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}' \qquad \llbracket E_2, \mathcal{M}'(E_1), \mathcal{M}' \rrbracket \rightarrow \mathcal{M}''}{\llbracket E_1 + E_2, \mathsf{S}, \mathcal{M} \rrbracket \rightarrow \mathcal{M}''' \cup \{E_1 + E_2 : \mathcal{M}'(E_1) \vee \mathcal{M}''(E_2)\}}$$

Figure 3.9. Syntax-directed rules for classification of expressions. Every expression in a program is assigned a shape, S, from our lattice, $\mathcal{L}$. This evaluation, denoted by $\llbracket E, \mathsf{S}, \mathcal{M} \rrbracket$, results in a new pair that is added to table $\mathcal{M}$ through $\mathcal{M} \cup \{E : \mathsf{S}\}$. This table is consulted during constraint generation of a binary expression, $E_1 \oplus E_2$, and of return-statements, as show in Figure 3.6. The rules in this figure mimic the restrictions imposed, by the C language, on operands of an expression. Even though it does not appear in this figure, we actually traverse the entire program (not only expressions) during the classification phase. In particular, information about declarations is gathered as well. For instance, upon `T* p;` we know that `T` must be a pointer. From a presentation standpoint, we focus on expressions, since they account for the most interesting rules. Our classification implementation can be found in Appendix A.10.

between α_1 and α_2, and to produce appropriate constraints: (i) for symmetric relations, an equivalence constraint, $\alpha_1 \equiv \alpha_2$, is produced; (ii) for subtyping relations, we produce constraint $\alpha_1 \leq \alpha_2$, which is further discussed in Section 4; (iii) if a relation cannot be asserted for the types in question, no constraint is produced – it is "dropped".

Table $\mathcal{M}$ is also consulted upon the use of the other auxiliary generator $\langle\langle\, \mathsf{S}_1, \alpha_1, \mathsf{S}_2, \alpha_2, \mathcal{M}, \oplus \,\rangle\rangle_{sel}$. This generator is called ***select***. Its purpose is to ensure that we take advantage of C's restrictions imposed on binary operations to refine the type of operands and the result of such expressions. To this end, additional type equivalence constraints are produced, based on the category of an operator. Let us discuss

$$\langle\langle\, \mathsf{S}_1, \alpha_1, \mathsf{S}_2, \alpha_2, \mathcal{M}, \oplus \,\rangle\rangle_{kd} = \begin{cases} \top, & \text{if } ((\mathsf{S}_1 \textbf{ ne } \mathsf{S}_2) \textbf{ and } (\mathsf{S}_1 \textbf{ is } p \textbf{ or } \mathsf{S}_2 \textbf{ is } p)) \\ \alpha_2 \leq \alpha_1, & \text{if } (\oplus \textbf{ is } =) \\ \alpha_1 \leq \alpha_2, & \text{if } (\mathsf{S}_1 \textbf{ is } i \textbf{ and } \mathsf{S}_2 \textbf{ is } fp) \\ \alpha_2 \leq \alpha_1, & \text{if } (\mathsf{S}_1 \textbf{ is } fp \textbf{ and } \mathsf{S}_2 \textbf{ is } i) \\ \alpha_1 \equiv \alpha_2 \end{cases}$$

$$\langle\langle\, \mathsf{S}_1, \alpha_1, \mathsf{S}_2, \alpha_2, \mathcal{M}, \oplus, \tau \,\rangle\rangle_{sel} = \begin{cases} \text{case } \oplus \text{ of} \\ \quad + \;\rightarrow \begin{cases} \alpha_1 \equiv \tau \wedge \alpha_2 \equiv \texttt{int}, & \text{if } (\mathsf{S}_1 \textbf{ is } p) \\ \alpha_2 \equiv \tau \wedge \alpha_1 \equiv \texttt{int}, & \text{if } (\mathsf{S}_2 \textbf{ is } p) \\ \tau \leq \texttt{double} \end{cases} \\ \quad = \;\rightarrow \tau \equiv \alpha_1 \\ \quad || \;\rightarrow \tau \equiv \texttt{int} \\ \quad / \;\rightarrow \begin{cases} \tau \equiv \texttt{int} \wedge \alpha_1 \equiv \texttt{int} \wedge \alpha_2 \equiv \texttt{int}, & \sharp \\ \tau \leq \texttt{double} \wedge \alpha_1 \equiv \texttt{int} \wedge \alpha_2 \leq \texttt{double}, & \flat \\ \tau \leq \texttt{double} \wedge \alpha_1 \leq \texttt{double} \wedge \alpha_2 \equiv \texttt{int}, & \natural \\ \tau \leq \texttt{double} \wedge \alpha_1 \leq \texttt{double} \wedge \alpha_2 \leq \texttt{double} \end{cases} \end{cases}$$

$$\sharp \text{ if } (\mathsf{S}_1 \textbf{ is } i \textbf{ and } \mathsf{S}_2 \textbf{ is } i) \qquad \flat \text{ if } (\mathsf{S}_1 \textbf{ is } i) \qquad \natural \text{ if } (\mathsf{S}_2 \textbf{ is } i)$$

Figure 3.10. The auxiliary generators *keep or drop* (*kd*), and *select* (*sel*). The former produces constraint between the operands E_1 and E_2 of a binary expression $E_1 \oplus E_2$, while the latter produces constraints that individually refine either E_1 or E_2, or the result type of the operation. In the conditions to the right of the case blocks, we use the textual notation **and**, **or**, **is**, and **ne** (*not equal*) to avoid confusion with operator symbols from μC. The implementation of these generators appears in Appendix A.11.

in more details each one of the auxiliary generators *keep or drop* and *select*. Their formal description appear in Figure 3.10; their implementation in Appendix A.11.

kd The *keep or drop* generator: Whenever one of the shapes involved in a relation is ranked as p in $\mathcal{L}$, and the other shape is an arithmetic one (*i.e.*, *n*, *i*, or *fp*), we say that this relation is *inconsistent*; otherwise, it is *consistent*. Upon a binary operation $E_1 \oplus E_2$, one would typically produce an equivalence constraint between the types of E_1 and E_2. Analogously, we would expect a relation of the same nature between the return type of a function and the return-statements that exist within it. However, a constraint associated with an inconsistent relation must be dropped - in fact, not generated at all; otherwise it would trigger an incompatible unification (called *over-unification* by Noonan *et al.* [Noonan et al., 2016]). Such dropping does not affect stability of our solver because there is no loss of information: given that both p and an arithmetic shape have high ranks, a constraint more restrictive than the one being dropped must exist. This fact is stated in Lemma 2.

Even upon consistent relations, we must still be cautious not to produce a constraint that would trigger undesired conversions, like the truncation of a `double` into an `int` or the assignment of a `const` to a non-`const` pointer. To deal with such subtleties, instead of producing a type equivalence constraint, $\alpha_1 \equiv \alpha_2$, the *keep or drop* generator may produce an inequality one. An inequality constraint, $\alpha_1 \leq \alpha_2$, implies a subtyping relation, which is necessary under the following circumstances: (i) upon matching a `return` E`;` to the return type of a function; (ii) always, for a binary expression that is an assignment, $E_1 = E_2$; (iii) whenever the sides of binary expressions, $E_1 + E_2$, E_1/E_2, and $E_1 \,||\, E_2$ are of different arithmetic shapes, *e.g.*, E_1 is *fp* and E_2 is *i*. Example 4 illustrates the role of *keep and drop*. The handling of subtyping relations by our constraint solver is discussed in Section 4.

sel The *select* generator: We have just presented *kd*, which produces a constraint (unless it is dropped) between the two operands of a binary expression. At this time, our goal is to produce a constraint that will individually refine either one of the operands or the result type of such operation. For an addition, +, as we have seen in Figure 3.9, if one of the operands is a pointer, then the other must be of integral type, *e.g.* an `int`; if both E_1 and E_2 are of arithmetic types, we know that the result is that of the highest ranked operand, which must be a subtype of `double`. The reasoning of this latter case, when both operands are arithmetic, apply to the division operation, /, as well. In a logical OR, the result type is always `int`, straight from the language rules. For assignments, *sel* ensures that the type of the overall expression is the same as that of the left-hand-side operand. This particular case is illustrated by Example 4 as well.

Lemma 2 (Classification of Expressions as Shapes)**.** *Given an expression E in a μC program P, if $\mathcal{M}(E)$ is* p *(respectively,* n*,* i*, or* fp*), then $\langle\langle\, P, \mathcal{M}\,\rangle\rangle_d$ generates constraints that bind E to a pointer (respectively, an arithmetic) type.*

Proof. Let us consider the case when $\mathcal{M}(E) = p$. One of the rules, Figure 3.9, that identifies a pointer, p, is $[\![\&E, \mathsf{S}, \mathcal{M}]\!]$. The construct $\&E$ is accordingly classified through $\mathcal{M} \cup \{\&E : p\}$. By inspection of the constraint generators in Figure 3.6,

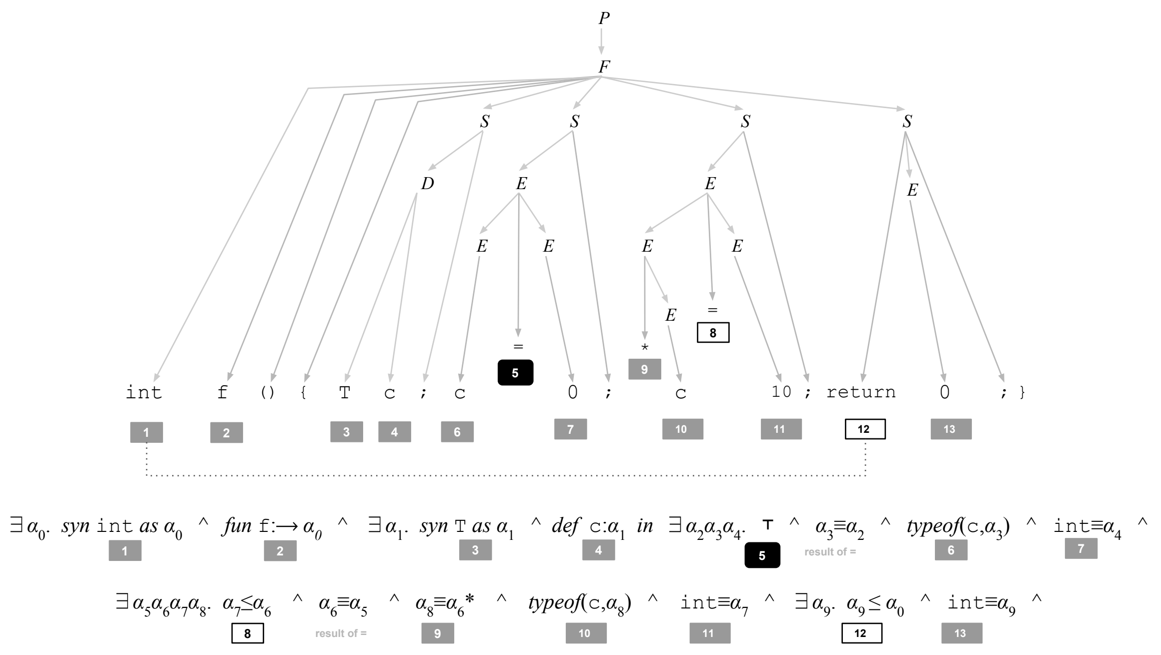

P
F
S
S
S
S
D
E
E
E
E
E
E
E
E
=
=
*
int f () { T c ; c 0 ; c 10 ; return 0 ; }
1 2 3 4 6 7 10 11 12 13
5
8
9
∃α0. syn int as α0 ∧ fun f:→ α0 ∧ ∃α1. syn T as α1 ∧ def c:α1 in ∃α2α3α4. ⊤ ∧ α3≡α2 ∧ typeof(c,α3) ∧ int≡α4 ∧
1 2 3 4 5 result of = 6 7
∃α5α6α7α8. α7≤α6 ∧ α6≡α5 ∧ α8≡α6* ∧ typeof(c,α8) ∧ int≡α7 ∧ ∃α9. α9≤α0 ∧ int≡α9 ∧
8 result of = 9 10 11 12 13

Figure 3.11. Constraint generation and the role of auxiliary generators *kd* and *sel*. This figure shows the constraints that are generated for the program in Figure 1.3 (c). Boxed numbers map constraints to the syntax responsible for producing them. We highlight a few cases: i) the black box numbered 5 illustrates the "dropping" of a constraint (*i.e.* it is not generated at all, and ⊤ stands for it). ii) the white boxes, 8 and 12, correspond to inequality constraints; those are remarkably relevant due to the way that our solver works.

$\langle\langle\, \& E : \tau, \mathcal{M}\,\rangle\rangle_e$, produces a type variable of pointer type, α_2*, that is equivalent to the type of E, which corresponds to α_1. The other rules that identify pointers or arithmetic shapes can be verified in a similar way. □

Example 4. An illustration of the constraints we generate for the program in Figure 1.3 (c) appears in Figure 3.11. We use a boxed number to map a constraint to the syntax that produces it. An inconsistent relation appears in the rounded-corner black box numbered 5. This constraint, associated with a binary expressions involving a pointer and an integral (`0`, the null pointer constant) gets dropped by the auxiliary generator *keep or drop*. Inequalities appear in the white boxes 8 and 12. The role of auxiliary generator *select* can be seen in the equivalences $\alpha_3 \equiv \alpha_2$ and $\alpha_6 \equiv \alpha_5$. In this case however, the return type of the assignment is discarded, given that it appears within an expression-statement.

A final consideration pertaining our lattice, $\mathcal{L}$, is that neither μC or C contains an actual `scalar_t` type. Therefore, if an expression remains with shape s, we could deliberately choose to make it an `int` (or any arithmetic or pointer type). This decision would not imply on any loss of generality. But care must be taken so that the chosen type is the same across all uses of the given expression. Furthermore, the language rules need to be respected, given that operations like addition are not defined for two pointer types, for instance. Toward a flexible approach, in PsycheC we generate a `typedef int scalar_t;` in this case.

3.5 The Type System of μC

A well typed μC program must respect the rules of Figure 3.12. There, Γ represents the typing context, and $\Gamma, x : \tau$ denotes the inclusion of x, typed as τ, into Γ. Lookup of the type bound to x within Γ is written as $\Gamma(x)$. The typing context is builtin with binary operators. Operands of certain expressions must be either of arithmetic type or of scalar type. To address this situation, we make use of unary predicates *ari* and *sc*. The former for either an `int` or `double`; the latter holds as well for those two types, and also for pointers. The definition of *ari* and *sc* appear in Figure 3.13 and their implementation in Appendix A.25.

The simplest typing of μC is that of a literal, *TLit*. In this case, the type can be asserted regardless of Γ. The other primary expression of our language is an identifier-expression, x, consisting of a program variable. Typing of a variable is done by rule *TVar*: it just looks up the τ associated with x in Γ. A variable can only be included into the typing context by either one of *TPar* or *TDclSt*. The only difference between *TPar* and *TDclSt* is that the former is applied on the formal parameters of a function, while the latter applies on declaration-statements.

Continuing with expressions, we now describe *TFld*. This rule first requires that the base expression E of a field access is typed. Based on the definition of μC, a pointer to a `struct`, $\tau_s *$, is to be expected. Then, we type the overall field access expression as τ_f, which is the type of x within the record τ_s. Pointer-related rules *TDrf* and *TAdr* are straightforward: they will construct or deconstruct a pointer type (informally, we can view one as the inverse of the other).

Let us now look into an assignment. We have split the typing of this expression into two rules. The reason for this separation is to accommodate the assignment of the *null pointer constant*, `0`, to both pointer and arithmetic types (a characteristic of C that has already been thoroughly discussed). Rule *TAsgZro* accounts for this situation. That is why, in the premise of this rule, it is enforced that τ is of scalar type. An assignment where the right-hand-side is non-zero is handled by *TAsg*. This

$$\frac{}{\Gamma \vdash \ell : \rho(\ell)}\ \{TLit\} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \{TVar\} \qquad \frac{\Gamma \vdash E : \tau_s{*} \quad \texttt{field}(x, \tau_s) = \tau}{\Gamma \vdash E\texttt{->}x : \tau}\ \{TFld\}$$

$$\frac{\Gamma \vdash E : \tau_p \quad \tau = \tau_p{*}}{\Gamma \vdash {*}E : \tau}\ \{TDrf\} \qquad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash \&E : \tau{*}}\ \{TAdr\}$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma \vdash E_s : \tau_s \quad \tau_s <: \tau}{\Gamma \vdash E = E_s : \tau}\ \{TAsg\} \qquad \frac{\Gamma \vdash E : \tau \quad sc(\tau)}{\Gamma \vdash E = \mathtt{0} : \tau}\ \{TAsgZr\}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad sc(\tau_1) \quad \Gamma \vdash E_2 : \tau_2 \quad sc(\tau_2) \quad \Gamma \vdash || : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{int}}{\Gamma \vdash E_1 \; || \; E_2 : \tau}\ \{TOr\}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad ari(\tau_1) \quad \Gamma \vdash E_2 : \tau_2 \quad ari(\tau_2) \quad \Gamma \vdash / : \tau_1 \rightarrow \tau_2 \rightarrow rank(\tau_1, \tau_2)}{\Gamma \vdash E_1 \; / \; E_2 : \tau}\ \{TDiv\}$$

$$\frac{\Gamma \vdash E_1 : \tau_1{*} \quad \Gamma \vdash E_2 : \texttt{int} \quad \Gamma \vdash +_{ptr-int} : \tau_1{*} \rightarrow \texttt{int} \rightarrow \tau_1{*}}{\Gamma \vdash E_1 + E_2 : \tau}\ \{TAddPtrInt\}$$

$$\frac{\Gamma \vdash E_1 : \texttt{int} \quad \Gamma \vdash E_2 : \tau_2{*} \quad \Gamma \vdash +_{int-ptr} : \texttt{int} \rightarrow \tau_2{*} \rightarrow \tau_2{*}}{\Gamma \vdash E_1 + E_2 : \tau}\ \{TAddIntPtr\}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad ari(\tau_1) \quad \Gamma \vdash E_2 : \tau_2 \quad ari(\tau_2) \quad \Gamma \vdash + : \tau_1 \rightarrow \tau_2 \rightarrow rank(\tau_1, \tau_2)}{\Gamma \vdash E_1 + E_2 : \tau}\ \{TAddAri\}$$

$$\frac{\Gamma, x : \tau \vdash \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma \vdash \texttt{\{}\tau\ x;\ S^\star\texttt{\}} : \tau_r}\ \{TDclSt\} \qquad \frac{\Gamma \vdash E : \tau \quad \Gamma \vdash \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma \vdash \texttt{\{}\,E;\ S^\star\texttt{\}} : \tau_r}\ \{TExpSt\}$$

$$\frac{\Gamma \vdash \texttt{(}D^\star\texttt{)}\,\texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma \vdash \tau_r\ f\texttt{(}D^\star\texttt{)}\,\texttt{\{}S^\star\texttt{\}} : \tau_r}\ \{TFun\} \qquad \frac{\Gamma, x : \tau \vdash \texttt{(}D^\star\texttt{)}\,\texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma \vdash \texttt{(}\tau\ x,\ D^\star\texttt{)}\,\texttt{\{}S^\star\texttt{\}} : \tau_r}\ \{TPar\} \qquad \frac{\Gamma \vdash \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma \vdash \texttt{()}\,\texttt{\{}S^\star\texttt{\}} : \tau_r}\ \{TBdy\}$$

$$\frac{\Gamma \vdash E : \tau \quad \tau <: \tau_r}{\Gamma \vdash \texttt{\{}\,\texttt{return}\ E;\texttt{\}} : \tau_r}\ \{TRet\} \qquad \frac{\Gamma \vdash \mathtt{0} : \tau_r \quad sc(\tau_r)}{\Gamma \vdash \texttt{\{}\,\texttt{return}\ \mathtt{0};\texttt{\}} : \tau_r}\ \{TRetZro\}$$

Figure 3.12. The typing rules for self-contained μC programs. $\Gamma, x : \tau$ means the introduction of x, with type τ, into Γ; correspondingly, $\Gamma(x)$ retrieves that type τ associated with x. Binary operators $+$, $/$, and $||$ are built into Γ. Functions *ari* and *sc* are predicates that hold for arithmetic and scalar types, respectively. `rank` is a function that selects the type with highest rank - in μC, we extend the concept of C's integer conversion rank to account for `double` as well: $rank(\texttt{double}, \texttt{int}) = \texttt{double}$ and $rank(\texttt{int}, \texttt{double}) = double$. We recall that, in μC, a `return` is always the last statement of a function.

latter rule independently types E_1 and E_2. It ensures that the subtyping relation introduced in Section 3.3, and further discussed in Section 4.1, is respected.

Binary expressions for division and logical OR share the same structure. Like C,

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
ari(\texttt{int}) & = & \top & sc(\texttt{int}) & = & \top \\
ari(\texttt{double}) & = & \top & sc(\texttt{double}) & = & \top \\
ari(\tau) & = & \bot & sc(\tau *) & = & \top \\
 & & & sc(\tau) & = & \bot
\end{array}
$$

Figure 3.13. Functions *ari* and *sc*. Both of them are unary predicates. The former is true, denoted by $\top$, when the argument passed is an arithmetic type; and false, denoted by $\bot$, otherwise. The latter is true for an arithmetic or a pointer type. Appendix A.25 shows the implementation of these functions.

our language requires that, in a division, both E_1 and E_2 are of arithmetic type; thus, we see the use of function *ari* in the premise of rule *TDiv*. In analogous manner, *TOr* enforces, by means of function *sc* in its premise, that the operands of disjunction are of scalar type. The only other variation among those two rules concerns their return type. *TOr* always return an `int`, while, for *TDiv*, we make use of an operator called `rank`. In C, the *rank* of an integer type depends on its precision [ISO-Standard, 2011]{§6.3.1.1}. But in μC, we slightly abuse this notion by incorporating a floating point type to this concept. In particular, the rank `double` is made higher than that of `int`.

The last expression to be discussed is addition. We have three rules for this operation. All of them share an underlying principle: they independently type E_1 and E_2, and, based on τ_1 and τ_2, the result type is determined. When the first operand of an additive expression is a pointer, then the second operand must be an `int`, and the result is of the same type as the E_1, *i.e.*, $\tau_1 *$. This reasoning corresponds to rule *TAddPtrInt*. The opposite situation, when the second operand is a pointer, has an analogous typing and is represented by rule *TAddIntPtr*. The third possible scenario of an addition happens when both of the operands are of arithmetic types. For this case, rule *TAddAri* applies and operator `rank` selects higher ranked type.

It now remains to discuss the following rules of Figure 3.12: *TExpSt*, *TRet*, *TRetZro*, *TFun*, and *TBdy*. The first one, *TExp*, applies on an expression-statement. We do not care about the type of E because it is thrown away. *TRet* and *TRetZro* handle the last statement of a function[12]. These two rules ensure that the type of a return-statement matches the enclosing function's declared return type, τ_r. Such association is made through rule *TCFun*. Note that μC does not contain syntax for invocations, so it is not necessary that we put f, the function name, into Γ. The role of *TBdy* is simply to propagate τ_r through the body until a `return` is found.

[12]We remind the reader that our formulation from Section 3.2 requires that functions have a valued return. In C, that is not the case, and PsycheC deals with non-valued returns by ignoring `void`

Chapter 4

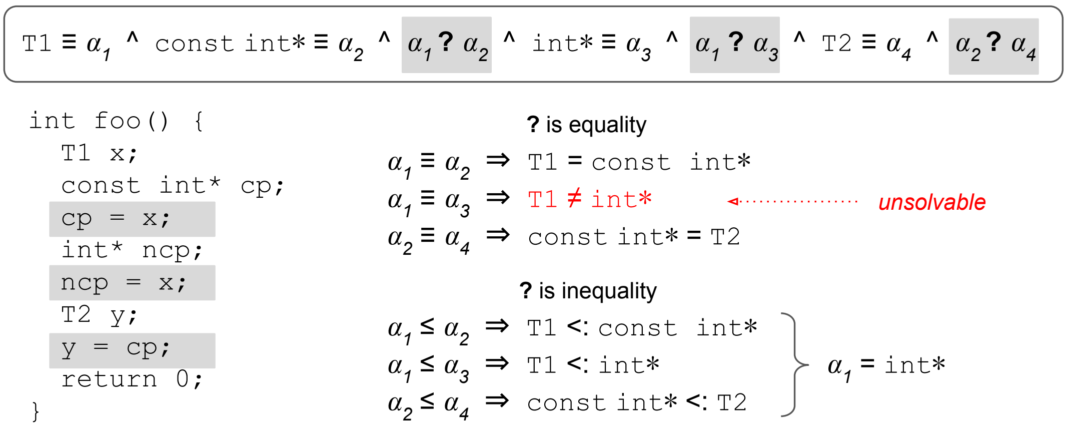
Translating Constraints into Types

To infer the types of a μC program, we must solve the constraints presented in Section 3.4. This procedure can be done via a first order logic model. Particularly often, using classical unification [Robinson, 1965]. However, the generators from Figure 3.6 produce constraints not only in the form of equivalences, but also of inequalities. That being said, the latter form of constraints prevents us from employing classical unification to our system as whole: both symmetric and asymmetric type relations exist.

As mentioned in Section 3.3, the behavior of pointer related conversions mimics that of a subtyping relation. a non-`const` pointer (in analogy, a subtype type), can be implicitly assigned to a `const` pointer (in analogy a base type), *i.e.*, $\tau* <: \texttt{const}\,\tau*$. But the reverse is not permitted. This modeling meets Liskov's substitution principle. Given such correspondence between subtyping and type qualifiers, we now rephrase the terminology introduced in Section 4 from *qualifier-neutral* and *qualifier-aware* to *subtyping-neutral* and *subtyping-aware*, respectively.

4.1 Subtyping and Unification

Let us exemplify how a naïve type inference system, one that does not account for asymmetries in type relations, would deal with the interaction among `const` and non-`const` pointers. In Figure 4.1, there is a program that features three assignments involving pointers. Above it, we show what would be the core constraints produced for such program. In order to illustrate the difference between equivalences and inequalities, we lave a question mark, **?**, in between the constraints that correspond to the assignments. If the **?** is made an $\equiv$, as it would be the case in classical unification, we end up with an unsolvable system (or we would need to indiscriminately discard the



```
int foo() {
  T1 x;
  const int* cp;
  cp = x;
  int* ncp;
  ncp = x;
  T2 y;
  y = cp;
  return 0;
}
```

$T1 \equiv \alpha_1 \wedge \text{const int*} \equiv \alpha_2 \wedge \alpha_1 ? \alpha_2 \wedge \text{int*} \equiv \alpha_3 \wedge \alpha_1 ? \alpha_3 \wedge T2 \equiv \alpha_4 \wedge \alpha_2 ? \alpha_4$
? is equality
$\alpha_1 \equiv \alpha_2 \Rightarrow$ T1 = const int*
$\alpha_1 \equiv \alpha_3 \Rightarrow$ T1 ≠ int*
unsolvable
$\alpha_2 \equiv \alpha_4 \Rightarrow$ const int* = T2
? is inequality
$\alpha_1 \leq \alpha_2 \Rightarrow$ T1 <: const int*
$\alpha_1 \leq \alpha_3 \Rightarrow$ T1 <: int*
$\alpha_2 \leq \alpha_4 \Rightarrow$ const int* <: T2
α_1 = int*

Figure 4.1. The effect of asymmetries on classical unification. At the top of the figure we show what would be typical constraints for the program to the left. Except that we use a question mark, ?, to represent the relation among types in an assignment - those are highlighted in gray. A conventional system that is based on classical unification would take the ? as $\equiv$, treating such relations as equivalence constraints. However, a consequence of this choice is that we are not able to unify `int*` with α_1 because, from $\alpha_2 \equiv \alpha_1$, we know that α_1 has already been instantiated as `const int*`. On the other hand, by having the ? as $\leq$, it is possible to find a solution for the given system of inequalities: because subtyping is a reflexive relation, `int*` <: `int*` holds; and for the partial order of pointer relations that we define, `int*` <: `const int*` holds as well.

`const` qualifier[1]). However, if the ? is replaced by $\leq$, a solution is possible, given that subtyping is reflexive and the interpretation of `int*` <: `const int*` is respected.

Unification, constraint-based type inference, and subtyping, in its different incarnations (atomic, structural, non-structural), have been studied by Kaes [Kaes, 1992], Smith [Smith, 1994], Pottier [Pottier, 1996, Pottier, 1998], Mitchell [Mitchell, 1991], Simonet [Simonet, 2003], and Su *et al.* [Su et al., 2002], among other researchers. However, in their systems, subtyping is treated with an extra set of constraints relations that need to be solved separately to the main inference algorithm. Recently, Dolan and Mycroft [Dolan and Mycroft, 2017] introduced *biunification*, and, as a means to deal with subtyping from first principles, they define *positive* types, τ^+, `negative` types, τ^-, and the notion of *bisubstitutions*. Yet powerful and elegant, their system comes with the extra complexity necessary to deal with bisubstitutions, which apply independently on each side of a predicate $\tau^+ \leq \tau^-$, resulting in an algebra of $\sqcap$ and $\sqcup$

[1]The kcc compiler [Ellison and Rosu, 2012, Hathhorn et al., 2015, Runtime-Verification, 2017], which strictly adheres to the C standard [ISO-Standard, 2011], rejects a program where `const`ness is lost; gcc and clang are more permissive: albeit with warnings, compilation succeeds - a further discussion about the behavior of C compilers appears in Section 5.

lattice operators solved alongside with unification. For our specific scenario, it possible to solve this problem without that extra complexity.

4.1.1 A Two-Phase Unification Approach

It may emerge as a hypothesis whether we could solve μC's constraints with an atomic subtyping system [Mitchell, 1991]. In fact, such approach would be similar to that of Foster *et al.* [Foster et al., 1999]. However, because the pointer declarator, $*$, is a type constructor that yields relations where the structure of the underlying type is of interest, atomic subtyping is not enough. Even though C's type system is a nominal one, this scenario indicates that structural inference rules should be used. Foster *et al.* [Foster et al., 1999] is solely concerned on inferring type qualifiers, not types entirely. To this end, they "lift" one level of type qualification in order to model the different behaviors of `const`: within and outside a pointer. But if we were to make use of only atomic subtyping in μC, we would either loose the ability to model `int` $<:$ `double` or `int`$*$ $<:$ `const int`$*$ and `double`$*$ $<:$ `const double`$*$; or we would be unable to detect the invalid relations `int`$*$ $<:$ `double`$*$ and `double`$*$ $<:$ `int`$*$.

Dolan and Mycroft's biunification, on the other hand, would be capable of solving the constraints generated for μC. Yet, we chose to develop a simpler algorithm for such task. To the best of our knowledge, this is the first approach that simultaneously deals with both type equivalence and inequality within a single solving method. Even though a type equivalence could be replaced by two type inequalities, the presence of the former lets us devise a simpler method to solve the latter. As opposed to biunification, the technique we develop does not require bisubstitutions; our method also gives importance to specific sides of a relation, though.

Our idea to deal with subtyping is to employ two different unification algorithms. The first one is classical unification; the second is an elaborated version of unification that is interleaved with a constraint *ordering* step which guarantees that the binding of a type variable participating in a weak relation only happens after the binding of any strong relations involving the type variable in question. Enforcing an specific partial order by which type variables are bound has the same effect of solving the constraints in any arbitrary sequence and later refining the types as according to such partial order. As in the system by Dolan and Mycroft, the side (left or right) of types in constraint relations must be preserved throughout the process. We refer to these two unification algorithms as $\mathcal{U}_c$ and $\mathcal{U}_s$. They are defined in Figure 4.2, with implementation in Appendix A.12. Both of them either halt due to an error or return a unifier. To denote the *trivial substitution*, we use `[]`.

$\mathcal{U}_c$ is the classical unification algorithm as presented by Martelli and Montanari [Martelli and Montanari, 1982]. It contains a type variable binding rule, a few rules that match ground types and eventually return an trivial substitution, and complementary rules that account for type destructuring.

$\mathcal{U}_s$ is a unification implementation that incorporates the subtyping-neutral and subtyping-aware strategies that we discussed before. The strategy is represented by the third parameter of the function. When the strategy value is not relevant, we denote it as _ (an underscore); otherwise, we use (i) `E`, when subtyping is to be **E**nforced; or (ii) `R`, when it is possible to **R**elax a subtyping requirement. To range over either `E` or `R`, simply passing the strategy to the next rule, we use m. By default, $\mathcal{U}_s$ starts at `R`. Whenever we identify that pointer types are being unified, the strategy is switched to `E`. During this state, conversions allowed by $\mathcal{U}_s$, such as the one from an `int` to a `double`, are forbidden. The state can be brought back to `R` if the type to the right (*i.e.*, the type on the left-hand-side of a μC assignment) is a `const` $\tau*$. An auxiliary *relax* function, defined in Figure 4.3 and available in Appendix A.13, is used during variable binding: depending on the context, it causes a possibly existing `const` qualifier to be discarded. The partial order resulting from $\mathcal{U}_s$ satisfies the relation, $\mathcal{R}_{\leq}$, below.

$$\mathcal{R}_{\leq} = \{\tau* \leq \texttt{const}\,\tau*, \texttt{double} \leq \texttt{int}, \alpha \leq \tau\}$$

Initially, unification $\mathcal{U}_c$ is applied on type equivalences. What emerges from this step is a primary type instantiation that, in spirit similar to that of Peyton Jones *et al.* [Peyton Jones et al., 2006], will serve as aid for the subsequent phases of our constraint solver. Later on, after *has* constraints are collected and grouped, unification $\mathcal{U}_c$ is once more used to ensure that different uses of a given field all have the same type. At this point, we are ready to start ordering inequalities, converting them into side-preserving type equivalences, and applying unification $\mathcal{U}_s$. The sorting phase guarantees that the order established by our subtyping criteria is respected. For instance, `const` $\tau* \leq \alpha$ would be handled before $\alpha \leq \tau*$, resulting in a safe binding order of type variables. Figure 4.4 revisits the same program that was previously shown in Figure 4.1, but now attention is focused on the effect that the ordering of inequalities has toward type instantiation. Further details about the role of our unification algorithms during the solving process are discussed in Section 4.2.

One may observe in the implementation from Figure 4.2 that no unification rules exist for types of the form `struct T` $\{D^*\}$. The reason for this absence is because

$$
\begin{array}{llll}
\mathcal{U}_c(\alpha, \tau) & = [\alpha \mapsto \tau] & \mathcal{U}_s(\alpha, \tau*, \_) & = [\alpha \mapsto \tau*] \\
\mathcal{U}_c(\tau, \alpha) & = \mathcal{U}_c(\alpha, \tau) & \mathcal{U}_s(\alpha, \tau, \mathtt{E}) & = [\alpha \mapsto \tau] \\
 & & \mathcal{U}_s(\alpha, \tau, \mathtt{R}) & = [\alpha \mapsto \mathit{relax}(\tau)] \\
 & & \mathcal{U}_s(\tau, \alpha, \_) & = [\alpha \mapsto \mathit{relax}(\tau)] \\
\mathcal{U}_c(\mathtt{int}, \mathtt{int}) & = \mathtt{[]} & \mathcal{U}_s(\mathtt{int}, \mathtt{int}, \_) & = \mathtt{[]} \\
 & & \mathcal{U}_s(\mathtt{int}, \mathtt{double}, \mathtt{R}) & = \mathtt{[]} \\
\mathcal{U}_c(\mathtt{double}, \mathtt{double}) & = \mathtt{[]} & \mathcal{U}_s(\mathtt{double}, \mathtt{double}, \_) & = \mathtt{[]} \\
\mathcal{U}_c(\mathtt{const}\ \tau_1, \mathtt{const}\ \tau_2) & = \mathcal{U}_c(\tau_1, \tau_2) & \mathcal{U}_s(\mathtt{const}\ \tau_1, \mathtt{const}\ \tau_2, m) & = \mathcal{U}_s(\tau_1, \tau_2, m) \\
 & & \mathcal{U}_s(\mathtt{const}\ \tau_1, \tau_2, \mathtt{R}) & = \mathcal{U}_s(\tau_1, \tau_2, \mathtt{R}) \\
\mathcal{U}_c(\tau_1*, \tau_2*) & = \mathcal{U}_c(\tau_1, \tau_2) & \mathcal{U}_s(\tau_1*, \tau_2*, \_) & = \mathcal{U}_s(\tau_1, \tau_2, \mathtt{E}) \\
\mathcal{U}_c(\mathcal{T}_{n_1}, \mathcal{T}_{n_2}) & = \mathtt{[]} \ \text{ if } n_1 {==} n_2 & \mathcal{U}_s(\mathcal{T}_{n_1}, \mathcal{T}_{n_2}, \_) & = \mathtt{[]} \ \text{ if } n_1 {==} n_2
\end{array}
$$

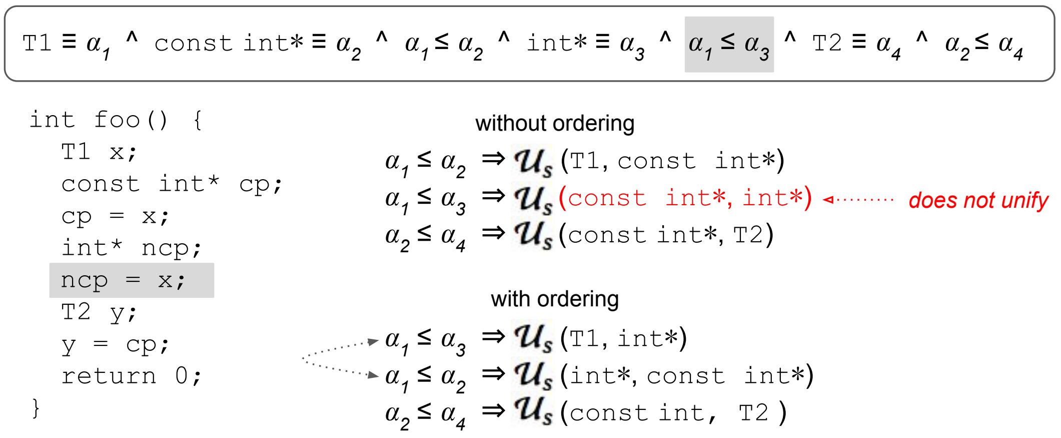
Figure 4.2. Unification algorithms $\mathcal{U}_c$ and $\mathcal{U}_s$. The relations allowed by each of the unification algorithms are not the same. Yet, they ensure that the type predicates from Figure 3.5 are respected. Function *relax*, used during type variable binding in $\mathcal{U}_s$, causes a `const` qualifier to be discarded, if one exists. Note the rule *PPtr* from Figure 3.5 is only triggered by $\mathcal{U}_c$, through the highlighted rule - as discussed, additional care is necessary for dealing with subtyping of pointers. That is the reason for the third parameter, the mode m: `R`elax or `E`nforce. Both $\mathcal{U}_c$ and $\mathcal{U}_s$ either halt due to an error or return a unifier. The implementation corresponding to this figure is available in Appendix A.12.

$$
\begin{array}{lcl}
\mathit{relax}(\mathtt{const}\ \tau) & = & \tau \\
\mathit{relax}(\tau*) & = & \mathit{relax}(\tau)* \\
\mathit{relax}(\tau) & = & \tau
\end{array}
$$

Figure 4.3. The *relax* function. "Relaxing" a type means to discard its `const` qualifier, if one exists. This function is used by unification algorithm $\mathcal{U}_c$. The implementation of *relax* appears in Appendix A.13.

records are *composed* from *has* constraints and, once that is achieved, their definitions are placed in Θ and only their typeids are relevant. As a result, unification of record types is indirectly done by their names. As explained in Section 3.3, this is a consequence of the fact that the μC (and C) adopts a *nominal* type system.

Another apparent omission in the implementation of our unification concerns types of the form $\tau \rightarrow \tau$. In this case, the reason why such rule does not appear in Figure 4.2 is because μC does not contain syntax for a function invocation. Therefore, unification between function types will never be triggered. Of course, this is not the case in C, and PsycheC's unification treat those appropriately. In fact, as we mention in Section 5, our type inference is capable of detecting variadic functions by catching errors that happen during unification of function arguments and formal parameters.



$T1 \equiv \alpha_1 \wedge \texttt{const int*} \equiv \alpha_2 \wedge \alpha_1 \leq \alpha_2 \wedge \texttt{int*} \equiv \alpha_3 \wedge \alpha_1 \leq \alpha_3 \wedge T2 \equiv \alpha_4 \wedge \alpha_2 \leq \alpha_4$

```
int foo() {
  T1 x;
  const int* cp;
  cp = x;
  int* ncp;
  ncp = x;
  T2 y;
  y = cp;
  return 0;
}
```

without ordering

$\alpha_1 \leq \alpha_2 \Rightarrow \mathcal{U}_s(\texttt{T1, const int*})$

$\alpha_1 \leq \alpha_3 \Rightarrow \mathcal{U}_s(\texttt{const int*, int*})$ does not unify

$\alpha_2 \leq \alpha_4 \Rightarrow \mathcal{U}_s(\texttt{const int*, T2})$

with ordering

$\alpha_1 \leq \alpha_3 \Rightarrow \mathcal{U}_s(\texttt{T1, int*})$

$\alpha_1 \leq \alpha_2 \Rightarrow \mathcal{U}_s(\texttt{int*, const int*})$

$\alpha_2 \leq \alpha_4 \Rightarrow \mathcal{U}_s(\texttt{const int, T2})$

Figure 4.4. The effect of inequalities ordering. This program is the same as the one in Figure 4.1. Again, we show, at the top, the constraints produced by our generators – already using $\leq$ for assignments. However, at this time, we focus attention to the order by which inequalities are passed to $\mathcal{U}_c$. As it can bee seen from the description of this algorithm in Figure 4.2, $\mathcal{U}_c$ does not accept a `const` pointer as the left operand and a non-`const` one as the right operand, unless the mode is currently set to Relax (which is not the case here, since we are dealing with pointer types). The absence of such rule corresponds to the inability of assigning a `const` pointer to a non-`const` one in μC and C. On the other hand, if we first bind the relation representing `ncp = x`, `T1` is instantiated as `int*`, and the second inequality goes through, since $\mathcal{U}_c$ admits the conversion `int*` <: `const int*`.

4.2 A Stage-based Solver

The syntax of constraints was presented in Section 3.2. There, we introduced *def* $x : \alpha$ *in* K, *typeof*(x, α), *has*$(\alpha, x : \tau)$, and, among others, $\tau_1 \equiv \tau_2$ and $\tau_1 \leq \tau_2$. Section 4.1 explains how the latter two varieties of constraints is solved. Yet, it remains to discuss how we handle those that are neither equivalences nor inequalities. This task, which we now give the details for, is accomplished through a *rewriting system.*

The *state* of our rewriting system is represented by means of a *configuration.* A configuration consists of a tuple formed by the maps ϕ, ψ, and Θ, a constraint K (produced by the generators from Section 3.4), and four auxiliary constraint lists. The solving process is a sequence of rewrites. At each iteration, constraints are either transformed or eliminated. The meaning of a configuration is preserved throughout the process. For instance, *typeof*(x, α) can be interpreted as $\tau_x \equiv \alpha$, where τ_x stands for the type of program variable x, as long as a corresponding side effect happens on ψ. As a matter of fact, this particular rewrite matches rule *KInst* from Figure 3.4. Appendix A.14 contains the implementation of our configuration, shown below.

$$\langle \phi, \psi, \Theta, K, [K_e], [K_i], [K_w], [K_f] \rangle$$

- K_e is a list of equivalences. A constraint can be of the form $\tau_1 \equiv \tau_2$ either because it was originally produced as such, or because it resulted from a configuration rewrite.
- K_i is a list of inequality constraints, *i.e.*, $\tau_1 \leq \tau_2$.
- K_w is a also list of inequalities, henceforth known as *wobbly* inequalities. The interpretation of a wobbly relation in our work is not the same as that of a wobbly type in the work of Peyton Jones *et al.* [Jones et al., 2004, Peyton Jones et al., 2006]. Yet, in both cases such term implies a notion of uncertainty in the type inference algorithm. A wobbly inequality is formalized by Definition 7 and illustrated in Example 5.
- K_f is a list of field membership constraints, *i.e.*, $\mathit{has}(\alpha, x : \tau)$.

Definition 7 (Wobbly Inequality)**.** A wobbly inequality is an inequality $\tau_1 \leq \tau_2$ where both τ_1 and τ_2 are type variables, *i.e.*, the constraint is of the form $\alpha_1 \leq \alpha_2$. $\diamond$

Example 5. None of the following inequalities are wobbly: $\alpha_1{*} \leq \alpha_2$, $\alpha_1{*} \leq \alpha_2{*}$, $\alpha_1 \leq \texttt{const}\ \alpha_2$, $\texttt{const}\ \alpha_1 \leq \texttt{const}\ \alpha_2$, $\alpha{*} \leq \texttt{double}$, $\mathcal{T}_n \leq \texttt{int}$, $\texttt{int} \leq \texttt{int}$.

Our solving process happens in stages. Each stage consists of a group of rules. We now present them, under the following names: i) preprocessing, ii) 1^{st} unification round, iii) 2^{nd} unification round, (iv) membership normalization, and (v) record composition. In Section 4.2.6, we discuss a subsequent stage for handling orphan type variables. The upcoming solver rules, use operator :: to prepend an element to a list, and operator ++ for list concatenation. For the application of a substitution σ over a list of constraints, $[K]$, and constraint group, $\mathcal{K}$, we augment the traditional notation by $\sigma[K]$ and $\sigma\mathcal{K}$, respectively. Because our mappings are help in an explicit way, by means of the data structures ϕ, ψ, and Θ, a *foreachValue* function is used for the application of a substitution over the types stored as values in those maps. A rewrite, written as $\rightsquigarrow$ with its name labeled above, is defined by the upcoming rules. The reflexive transitive closure of a rewrite is written as $\rightsquigarrow *$.

4.2.1 Preprocessing

Rules from this group, illustrated in Figure 4.5, are the first ones to be applied during the solving process. *PP-and* ensures that each constraint of a conjunction is preprocessed. *PP-ex* inserts an identity mapping into ϕ - in the semantics of Figure 3.4, *KEx* expects that such α will eventually be instantiated. *PP-syn* guarantees that a type variable is matched with the *canonical* version of a type: If the computed typeid is known, an equivalence constraint is created; otherwise, a new mapping in Θ is added. Both *PP-def* and *PP-fun* introduce x into ψ (the former needs to preprocess the enclosed constraint as well). The difference between those and *PP-inst* is that the latter assumes that a program identifier is already known. The last three remaining rules, *PP-has*, *PP-eq*, and *PP-iq* simply move the current constraint to the slot in the configuration that is reserved for that kind of constraint: $has(\alpha, x : \tau)$ goes to $[K_f]$; $\tau_1 \equiv \tau_2$ goes to $[K_e]$; and $\tau_1 \leq \tau_2$ is moved to $[K_i]$. In summary, the preprocessing rules simply move constraints around or eliminates them with a side effect on ϕ, ψ, and Θ. This observation allows to state Lemma 3. There, and in the remainder of this text, Θ_{std} is the map of *standard* types, containing pairs of typeids and definitions for builtins like `int` and `double`. The implementation of the preprocessing stage appears in Appendix A.15.

Lemma 3 (Termination of Preprocessing Stage). *Let* $\mathcal{C}^{\triangle} = \langle \phi^{\triangle}, \psi^{\triangle}, \Theta^{\triangle}, K, [K_e]^{\triangle}, [K_i]^{\triangle}, [K_w]^{\triangle}, [K_f]^{\triangle} \rangle$ *be any configuration assigned to* K. *By rewriting* $\mathcal{C}^{\triangle}$ *with* $\overset{PP}{\rightsquigarrow}$, *as defined in Figure 4.5, either the solver halts due to an error, or* $\exists \mathcal{C}_{pp} = \langle \phi_{pp}, \psi_{pp}, \Theta_{pp}, \top, [K_e]_{pp}, [K_i]_{pp}, [K_w]_{pp}, [K_f]_{pp} \rangle$, *such that* $\mathcal{C}^{\triangle} \overset{PP}{\rightsquigarrow}* \ \mathcal{C}_{pp}$.

Proof. By strong induction on the size of a constraint K. We show that, at each iteration, if the solver does not halt due to an error, then we are taken, by one or two steps, to a configuration where either K decreases or K is $\top$. Thus, we eventually arrive at a preprocessed form, $\langle \phi, \psi, \Theta, \top, [K_e], [K_i], [], [K_f] \rangle$. During the $\overset{PP}{\rightsquigarrow}$ rewrite, side effects on $\phi, \psi, \Theta, [K_e], [K_f]$, or $[K_i]$ do not influence termination. The size of a constraint is defined as follows.

$$
\begin{array}{lcl}
\mathsf{size}(\top) & = & 1 \\
\mathsf{size}(\bot) & = & 1 \\
\mathsf{size}(\tau_1 \equiv \tau_2) & = & 1 \\
\mathsf{size}(\tau_1 \leq \tau_2) & = & 1 \\
\mathsf{size}(has(\alpha, x : \tau)) & = & 1 \\
\mathsf{size}(typeof(x, \tau)) & = & 1 \\
\mathsf{size}(syn\, \tau\, \alpha) & = & 1 \\
\mathsf{size}(\exists \alpha . K) & = & 1 + \mathsf{size}(K) \\
\mathsf{size}(def\ x : \alpha\, in\, K) & = & 1 + \mathsf{size}(K) \\
\mathsf{size}(fun\, f : \tau \to \alpha\, in\, K) & = & 1 + \mathsf{size}(K) \\
\mathsf{size}(K_1 \wedge K_2) & = & \mathsf{size}(K_1) + \mathsf{size}(K_2)
\end{array}
$$

$\{PP\!-\!and\}$
$\langle \phi, \psi, \Theta, K_1 \wedge K_2, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi'', \psi'', \Theta'', \top, [K_e]'', [K_i]'', [], [K_f]'' \rangle$
`where`
$\langle \phi, \psi, \Theta, K_1, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$
$\langle \phi', \psi', \Theta', K_2, [K_e]', [K_i]', [], [K_f]' \rangle \overset{PP}{\rightsquigarrow} \langle \phi'', \psi'', \Theta'', \top, [K_e]'', [K_i]'', [], [K_f]'' \rangle$

$\{PP\!-\!ex\}$
$\langle \phi, \psi, \Theta, \exists \alpha.K, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$
`where`
$\langle \phi \cup \{\alpha, \alpha\}, \psi, \Theta, K, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$

$\{PP\!-\!def\}$
$\langle \phi, \psi, \Theta, def\ x : \alpha\ in\ K, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$
`where`
$\langle \phi, \psi \cup \{x, \phi(\alpha)\}, \Theta, K, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$

$\{PP\!-\!fun\}$
$\langle \phi, \psi, \Theta, fun\ f : \tau \to \alpha\ in\ K, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$
`where`
$\langle \phi, \psi \cup \{f, \tau \to \phi(\alpha)\}, \Theta, K, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]' \rangle$

$\{PP\!-\!syn\}$
$\langle \phi, \psi, \Theta, syn\ \tau\ as\ \alpha, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi, \psi, \Theta', \Theta'(\widehat{\tau}) \equiv \alpha, [K_e], [K_i], [], [K_f] \rangle$
`where`
$\Theta' =$ `if` $\widehat{\tau} \in \Theta$ `then` Θ `else` $\Theta \cup \{\widehat{\tau}, \alpha\}$

$\{PP\!-\!inst\}$
$\langle \phi, \psi, \Theta, typeof(x, \tau), [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi, \psi, \Theta, \psi(x) \equiv \tau, [K_e], [K_i], [], [K_f] \rangle$ `if` $x \in \psi$

$\{PP\!-\!has\}$
$\langle \phi, \psi, \Theta, has(\alpha, x : \tau), [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [K_e], [K_i], [], has(\alpha, x : \tau) :: [K_f] \rangle$

$\{PP\!-\!eq\}$
$\langle \phi, \psi, \Theta, \tau_1 \equiv \tau_2, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, (\tau_1 \equiv \tau_2) :: [K_e], [K_i], [], [K_f] \rangle$

$\{PP\!-\!iq\}$
$\langle \phi, \psi, \Theta, \tau_1 \leq \tau_2, [K_e], [K_i], [], [K_f] \rangle \overset{PP}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [K_e], (\tau_1 \leq \tau_2) :: [K_i], [], [K_f] \rangle$

Figure 4.5. The preprocessing rules of the solving process. Rule *PP-ex* is responsible for populating ϕ with existentially quantified type variables (it uses an identity mapping for that purpose). In this stage, the solver only halts due to an error if a malformed K is detected by *PP-inst*. None of these rules add constraints to $[K_w]$, that is why we show it as an empty list. The implementation of preprocessing appears in Appendix A.15.

– Basis: $\mathsf{size}(K) = 1$. The constraint must be one of $\top$, $\bot$, $has(\alpha, x : \tau)$, $\tau_1 \equiv \tau_2$, $\tau_1 \leq \tau_2$, $typeof(x, \tau)$, or $syn\, \tau\, \alpha$.

  - $\top$: By *PP-*$\top$, the configuration is simply propagated, it is already a preprocessed one.
  - $\bot$: The solver halts due to an error, given no rule matches this constraint.
  - $\tau_1 \equiv \tau_2$, $\tau_1 \leq \tau_2$, and $has(\alpha, x : \tau)$: By *PP-eq*, *PP-iq*, and *PP-has*, respectively, we take a step to a configuration where K becomes $\top$. An error never happens.
  - $typeof(x, \tau)$: By *PP-inst*, either the solver halts due to an error (*i.e.*, in the event that $x \in \psi$ does not hold), or we take a step to a configuration where K becomes an equivalence constraint. By *PP-eq*, a subsequent step is taken a to a preprocessed configuration.
  - $syn\, \tau\, \alpha$: By *PP-syn*, by an argument similar to the one used to deal with $typeof(x, \tau)$. The difference is that an error never happens in this case.

– Induction: Let us assume that our claim holds for $\mathsf{size}(K) \leq \mathcal{N}$, for any $\mathcal{N} > 1$. We show that it continues to hold for $\mathsf{size}(K) = \mathcal{N} + 1$. In such setup, K must contain, within it, at least a conjunction, $K_1 \wedge K_2$, or one of recursive constraints $\exists\alpha.K$, $def\ x : \alpha\ in\ K$, and $fun\ f : \tau \to \alpha\ in\ K$.

  - $K_1 \wedge K_2$: We have that $\mathsf{size}(K_1 \wedge K_2) = \mathcal{N} + 1 \Rightarrow \mathsf{size}(K_1) + \mathsf{size}(K_2) - \mathcal{N} = 1$. Since no constraint of size 0 exists, both $\mathsf{size}(K_1) \leq \mathcal{N}$ and $\mathsf{size}(K_2) \leq \mathcal{N}$. By the hypothesis, either the solver halts due to an error or $\langle\phi, \psi, \Theta, K_1, [K_e], [K_i], [], [K_f]\rangle \overset{PP}{\rightsquigarrow} \langle\phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]'\rangle$. Analogous reasoning can be employed for K_2.
  - $def\ x : \alpha\ in\ K$, $fun\ f : \tau \to \alpha\ in\ K$, or $\exists\alpha.K$: The proofs are similar for these three constraint forms. For the demonstration, we use the first one. There, $def\ x : \alpha\ in\ K = \mathcal{N} + 1 \Rightarrow 1 + \mathsf{size}(K) = \mathcal{N} + 1 \Rightarrow \mathsf{size}(K) = \mathcal{N}$. By the hypothesis, either the solver halts due to an error or $\langle\phi, \psi \cup \{x, \phi(\alpha)\}, \Theta, K, [K_e], [K_i], [], [K_f]\rangle \overset{PP}{\rightsquigarrow} \langle\phi', \psi', \Theta', \top, [K_e]', [K_i]', [], [K_f]'\rangle$.

$\square$

Another fact about the preprocessing rules is that they preserve the semantics of constraints from Figure 3.4. To establish this property, we adopt the notion of *entailment*. But, as opposed to Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005], we do not do it on the constraint level: instead, our statements are at the configuration level. The explanation for such deviation is because we reason about constraints alongside ϕ, ψ, and Θ. One configuration entails another, as formalized in Definition 8, if the former is at least as strict as the latter. There, $\mathcal{K}$ is the constraint *group* consisting of K, $[K_e]$, $[K_i]$, $[K_w]$, and $[K_f]$. In Lemma 4, we claim that preprocessing is stable in regards to the semantics of constraints.

Definition 8 (Configuration Entailment)**.** A configuration $\mathcal{C}$ entails a configuration $\mathcal{C}'$, denoted $\mathcal{C} \Vdash \mathcal{C}'$, if and only if, for any ϕ, ψ, and Θ, the assertion ϕ, $\psi, \Theta \models K$, $\forall K \in \mathcal{K}$ $\Rightarrow \phi, \psi, \Theta \models K'$, $\forall K' \in \mathcal{K}'$, where $\mathcal{K}$ is the constraint group of $\mathcal{C}$, and $\mathcal{K}'$ that of $\mathcal{C}'$. We write $\mathcal{C} \equiv \mathcal{C}'$ if and only if $\mathcal{C} \Vdash \mathcal{C}'$ and $\mathcal{C}' \Vdash \mathcal{C}$ holds. ◇

Lemma 4 (Stability of Preprocessing Stage)**.** *Let* $\mathcal{C}^{\triangle} = \langle \phi^{\triangle}, \psi^{\triangle}, \Theta^{\triangle}, K, [K_e]^{\triangle}, [K_i]^{\triangle}, [K_w]^{\triangle}, [K_f]^{\triangle} \rangle$ *be any configuration assigned to* K*. If* $\exists \mathcal{C}_{pp} = \langle \phi_{pp}, \psi_{pp}, \Theta_{pp}, \top, [K_e]_{pp}, [K_i]_{pp}, [K_w]_{pp}, [K_f]_{pp} \rangle$*, such that* $\mathcal{C}^{\triangle} \overset{PP}{\rightsquigarrow}* \ \mathcal{C}_{pp}$*, then* $\mathcal{C}^{\triangle} \equiv \mathcal{C}_{pp}$*.*

Proof. By case analysis on the rules composing the $\overset{PP}{\rightsquigarrow}$ rewrite, from Figure 4.5. We refer to the constraint group of $\mathcal{C}^{\triangle}$ as $\mathcal{K}^{\triangle}$, and to that of $\mathcal{C}_{pp}$ as $\mathcal{K}_{pp}$.

Case *PP-has*, *PP-eq*, and *PP-iq.*

These rules simply move K to one of the auxiliary lists $[K_e]$, $[K_i]$, or $[K_f]$. Thus, $\mathcal{K}^{\triangle} = \mathcal{K}_{pp}$ and $\mathcal{C}^{\triangle} \equiv \mathcal{C}_{pp}$ immediately holds.

Case *PP-inst.*

This rewrites *typeof*(x, τ) as $\psi(x) \equiv \alpha$. By *KInst*, from Figure 3.4, $\mathcal{C}^{\triangle} \equiv \mathcal{C}_{pp}$ holds.

Case *PP-def* and *PP-fun.*

PP-def inserts the pair $\{x, \phi(\alpha)\}$ in ψ, ensuring that the mapping $\psi[x \mapsto \phi(\alpha)]$ exists, and forwards the nested K. By *KDef*, from Figure 3.4, $\mathcal{C}^{\triangle} \equiv \mathcal{C}_{pp}$ holds. Analogous reasoning can be employed for *PP-fun* and *KFun.*

Case *PP-ex.*

The same as *PP-def* and *PP-fun*, but the map where the insertion happens is ϕ. By *KEx*, from Figure 3.4, $\mathcal{C}^{\triangle} \equiv \mathcal{C}_{pp}$ holds.

Case *PP-syn.*

This either inserts or updates the pair $\{\widehat{\tau}, \alpha\}$ in Θ, ensuring that the mapping $\Theta[\widehat{\tau} \mapsto \tau']$ exists, and replaces K with $\Theta(\widehat{\tau}) \equiv \alpha$. By *KSyn*, from Figure 3.4, $\mathcal{C}^{\triangle} \equiv \mathcal{C}_{pp}$ holds.

Case *PP-and.*

Similar to the proof of Lemma 3. □

4.2.2 1^{st} Unification Round

Once preprocessing is over, we start a sequence of stages that bind type variables. Type equivalences are the first constraints to be addressed. The relations in $[K_e]$ have either been produced as such by our generators from Section 3.4, or they resulted from a rewrite transformation that happened during the preprocessing stage. The rules of this group appear in Figure 4.6. *UE-base* unifies both sides of an equivalence via unification algorithm $\mathcal{U}_c$, from Figure 4.2. At each iteration, if such a call does not halt due to an error, then a substitution σ is returned. If σ is not the trivial substitution, then it is a unifier $[\alpha \mapsto \tau]$ that is *applied* to maps ϕ, ψ, and Θ, and to constraint lists $[K_e]$, $[K_i]$, and $[K_w]$ – note that it is not necessary modify $[K_w]$ because, at this moment, it is empty. To express the correlation between the maps ϕ, ψ, and Θ, to the maps ϕ', ψ', and Θ' of a rewritten configuration, we introduce Definition 9; likewise, Definition 10 states a similar correlation between $\mathcal{K}$ and $\mathcal{K}'$. The effect of applying a substitution σ, that is a unifier, on the elements of a configuration is formalize by Lemma 5. The implementation of the 1^{st} unification round appears in Appendix A.16.

Definition 9 (Mapping Extension)**.** Let $\mathcal{D}$ be any of the ϕ, ψ, or Θ, and $\mathcal{D}'$ be a respective ϕ', ψ', or Θ'. If $dom(\mathcal{D}) = dom(\mathcal{D}')$, and $\forall \mathsf{e} \in dom(\mathcal{D})$, relations $\mathcal{D}(\mathsf{e}) <: \tau$ and $\mathcal{D}'(\mathsf{e}) <: \tau'$, where $\tau <: \tau'$, hold, then we say that $\mathcal{D}'$ is an extension of $\mathcal{D}$, denoted by $\mathcal{D} \preceq \mathcal{D}'$. $\diamond$

Definition 10 (Constraint Reduction)**.** Let $\mathcal{K}$ be a constraint group, such that $\tau_1 \equiv \tau_2$ (alternatively, $\tau_1 \leq \tau_2$) $\in \mathcal{K}$, and $\mathcal{K}'$ be another constraint group, such that $\mathcal{K}' = \mathcal{K} \setminus \tau_1 \equiv \tau_2$ (alternatively, $\tau_1 \leq \tau_2$). If $\forall K \in \mathcal{K}'$, $\tau_1 = \tau_2$, then we say that $\mathcal{K}'$ is a reduction of $\mathcal{K}$, denoted by $\mathcal{K} \succeq \mathcal{K}'$. $\diamond$

Lemma 5 (Extension of ϕ and Reduction of $\mathcal{K}$ under Substitution)**.** *Let* $\langle \phi, \psi, \Theta, \top, [K_e], [K_i], [K_w], [K_f] \rangle$ *be a configuration, such that* $K = \tau_1 \equiv \tau_2$ *(alternatively,* $\tau_1 \leq \tau_2$*)* $\in \mathcal{K}$*. If* $\sigma = \mathcal{U}_c(\tau_1, \tau_2)$ *(alternatively* $\mathcal{U}_s(\tau_1, \tau_2)$*), then, foreachValue*$(\sigma,\ \phi) = \phi' \Rightarrow \phi \preceq \phi'$*, and* $\sigma\,(\mathcal{K} \setminus K) = \mathcal{K}' \Rightarrow \mathcal{K} \succeq \mathcal{K}'$*.*

Proof. When σ is the trivial substitution, $[]$, the proof is immediate, since $\phi = \phi' \Rightarrow \phi \preceq \phi'$, and $\mathcal{K} \setminus K = \mathcal{K}' \Rightarrow \mathcal{K} \succeq \mathcal{K}'$. Otherwise, σ is a unifier $[\alpha_\sigma \mapsto \tau_\sigma]$ for $\tau_1 \equiv \tau_2$. The part $\mathcal{K} \succeq \mathcal{K}'$ is straightforward. For the mappings, applying the substitution leads to one of the following, depending on the structure of τ_σ:

– $\tau_\sigma = \alpha_{\sigma'}$: The predicates involving a type such that $\phi(\alpha) = \alpha_\sigma = \tau$ hold both in ϕ and ϕ' by *SVrv*, *SVlv*, *PVrv*, and *PVlv*, possibly through a different type variable. Predicates involving a type τ such that $ftv(\tau) = \alpha_\sigma$ continue to hold in ϕ' by transitivity.

– $ftv(\tau_\sigma) = \varnothing$: $\forall \alpha \in \phi$, where $\phi(\alpha) = \alpha_\sigma = \tau$, we have that $\tau <: \tau_1$ and $\tau <: \tau_2$ (or $\tau_1 <: \tau$ and $\tau_2 <: \tau$). Because σ is an unifier, $\tau_\sigma <: \tau_1$ and $\tau_\sigma <: \tau_2$ (or

$\{\mathit{UE{-}base}\}$

$\langle \phi, \psi, \Theta, \top, (\tau_1 \equiv \tau_2) :: [K_e], [K_i], [], [K_f] \rangle \overset{\scriptscriptstyle UE}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [], [K_i]', [K_f]' \rangle$

```
  where
  $\sigma = \mathcal{U}_c(\tau_1, \tau_2)$
  $\phi' = \mathit{foreachValue}(\sigma, \phi)$
  $\psi' = \mathit{foreachValue}(\sigma, \psi)$
  $\Theta' = \mathit{foreachValue}(\sigma, \Theta)$
  $[K_e]' = \sigma[K_e]$
  $[K_i]' = \sigma[K_i]$
  $[K_f]' = \sigma[K_f]$
```

Figure 4.6. Rules for the 1st unification round of the solving process. $\mathcal{U}_c$ is essentially the classical unification algorithm. The substitutions obtained are applied the maps of ϕ, ψ, and Θ, and to the constraints in the auxiliary lists, except for $[K_w]$, which is empty. Appendix A.16 shows the implementation of this stage.

$\tau_1 <: \tau_\sigma$ and $\tau_2 <: \tau_\sigma$) hold in ϕ'. This means that certain relations that hold, in ϕ, by *SVrv*, *SVlv*, *PVrv*, and *PVlv*, now hold by *SVr*, *SVl*, *PVr*, and *PVl* in ϕ'. Predicates that transitively rely on those continue to hold.

$\square$

Corollary 2 (Extension of ψ and Θ through ϕ)**.** *Let σ be a substitution. If $\mathit{foreachValue}(\sigma, \phi) = \phi' \Rightarrow \phi \preceq \phi'$, then, $\mathit{foreachValue}(\sigma, \psi) = \psi' \Rightarrow \psi \preceq \psi'$, and $\mathit{foreachValue}(\sigma, \Theta) = \Theta' \Rightarrow \Theta \preceq \Theta'$.*

Proof. The types in ψ and Θ are just a mirror of the types in ϕ. $\square$

4.2.3 2nd Unification Round

Before the rules from this group are applied, we have already instantiated types whose relations derive from declarations present in a program; from associations with literals; or from binary expressions (when they can be determined, through our lattice from Section 3.4). It is now time to instantiate types that are modeled through subtyping relations, like those in assignments. To deal with such inequality constraints, we employ the second algorithm from Figure 4.2: unification $\mathcal{U}_s$. However, prior to doing so, we must order $[K_i]$ and separate *wobbly* inequalities from non-wobbly ones. To this end, we rely on supporting functions *splitWob* and *orderSub*, defined in Figure 4.7, and on rewrite *SO*.

$$
\begin{aligned}
&\textit{splitWob}\;(\;(\,\alpha_1 \leq \alpha_2\,) :: [K_1], [K_2]\,) = \textit{splitWob}\,(\,[K_1], (\,\alpha_1 \leq \alpha_2\,) :: [K_2]\,)\\
&\textit{splitWob}\;(\;(\,\tau_1 \leq \tau_2\,) :: [K_1], [K_2]\,) = \textit{splitWob}\,(\,[K_1]{+}{+}(\,\tau_1 \leq \tau_2\,), [K_2]\,)\\
&\textit{splitWob}\;(\;\bot :: [K_1],\;[K_2]\;) = [K_1], [K_2]
\end{aligned}
$$

$$
\begin{aligned}
&\textit{orderSub}\;(\;(\texttt{const}\;\tau_1{*} \leq \tau_2) :: [K_k],\;[K_s]\;) = \textit{orderSub}\;(\;[K_k],\;(\texttt{const}\;\tau_1{*} \leq \tau_2) :: [K_s]\;)\\
&\textit{orderSub}\;(\;(\tau_1 \leq \tau_2{*}) :: [K_k],\;[K_s]\;) = \textit{orderSub}\;(\;[K_k],\;(\tau_1 \leq \tau_2{*}) :: [K_s]\;)\\
&\textit{orderSub}\;(\;(\texttt{double} \leq \tau) :: [K_k],\;[K_s]\;) = \textit{orderSub}\;(\;[K_k],\;(\texttt{double}\;\leq \tau) :: [K_s]\;)\\
&\textit{orderSub}\;(\;(\tau \leq \texttt{int}) :: [K_k],\;[K_s]\;) = \textit{orderSub}\;(\;[K_k],\;(\tau \leq \texttt{int}) :: [K_s]\;)\\
&\textit{orderSub}\;(\;(\tau_1 \leq \tau_2) :: [K_k],\;[K_s]\;) = \textit{orderSub}\;(\;[K_k]\;{+}{+}\;[(\tau_1\;\leq \tau_2)], [K_s])\\
&\textit{orderSub}\;(\;\bot :: [K_k],\;[K_s]\;) = [K_s]\;{+}{+}\;[K_k]
\end{aligned}
$$

Figure 4.7. Function *splitWob*, responsible for the splitting of wobbly relations (returning a tuple with separate lists), and function *orderSub*, which ensures that subtyping is respected during variable binding, as discussed in Section 4.1. Constraint $\bot$ plays the role of a sentinel in both functions. In *orderSub*, list $[K_s]$ stands for *strong* constraint requirements. Those take precedence (*i.e.*, must be unified before) over *weak* ones, represented by list $[K_k]$. The implementation of these supporting functions is available in Appendix A.17.

The rationale for such splitting is that a relation like $\alpha_1 \leq \alpha_2$ does not refine a type on neither of its sides. Non-wobbly inequalities, on the other hand, should be interpreted as an strengthened constraint that resulted from the 1^{st} unification round. To ensure a proper ordering of type variable binding, we consider such inequalities with higher "priority". Therefore, they are unified first, through rule *UI-base*. Subsequently, *UW-base* proceeds with the unification of (eventually remaining) wobbly relations. One way to reason about this approach is as follows: Wobbly relations propagate, but do not influence, a type's requirements. Once they are are unified, no subtyping relation will be violated. Both *splitWob* and *orderSub*, terminate, as stated in Lemma 6. The implementation of the rules from the 2^{st} unification round appear in Appendix A.18.

Lemma 6 (Termination of Inequality Ordering and Wobbly Splitting)**.** *If any of the functions orderSub and splitWob, defined in Figure 4.7, is invoked with an argument $[K]$, whose last element (i.e., the sentinel) is the constraint $\bot$, then the function returns.*

Proof. Both functions rely on a stop condition determined by a sentinel, whose distance to the head of the input list decreases at each step. Eventually, $\bot$ is matched and the function returns. □

$\{SO\}$

$\langle \phi, \psi, \Theta, \top, [], [K_i], [], [K_f] \rangle \overset{SO}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [], [K_i]'', [K_w], [K_f] \rangle$

`where`

$[K_i]', [K_w] = \mathit{splitWob}([K_i] \mathbin{+\!+} [\bot])$

$[K_i]'' = \mathit{orderSub}\ ([K_i]' \mathbin{+\!+} [\bot], [])$

$\{UI{-}base\}$

$\langle \phi, \psi, \Theta, \top, [], (\tau_1 \leq \tau_2) :: [K_i], [K_w], [K_f] \rangle \overset{UI}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [], [K_i]''', [K_w]'', [K_f]' \rangle$

`where`

$\sigma = \mathcal{U}_s(\tau_1, \tau_2)$

$\phi' = \mathit{foreachValue}(\sigma, \phi)$

$\psi' = \mathit{foreachValue}(\sigma, \psi)$

$\Theta' = \mathit{foreachValue}(\sigma, \Theta)$

$[K_i]' = \sigma[K_i]$

$[K_w]' = \sigma[K_w]$

$[K_f]' = \sigma[K_f]$

$[K_i]'', [K_w]'' = \mathit{splitWob}([K_i]' \mathbin{+\!+} [K_w]', [\bot])$

$[K_i]''' = \mathit{orderSub}\ ([K_i]'' \mathbin{+\!+} [\bot], [])$

$\{UW{-}base\}$

$\langle \phi, \psi, \Theta, \top, [], [], (\tau_1 \leq \tau_2) :: [K_w], [K_f] \rangle \overset{UW}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [], [], [K_w]', [K_f]' \rangle$

`where`

$\sigma = \mathcal{U}_s(\tau_1, \tau_2)$

$\phi' = \mathit{foreachValue}(\sigma, \phi)$

$\psi' = \mathit{foreachValue}(\sigma, \psi)$

$\Theta' = \mathit{foreachValue}(\sigma, \Theta)$

$[K_w]' = \sigma[K_w]$

$[K_f]' = \sigma[K_f]$

Figure 4.8. Rules for the 2^{nd} unification round of the solving process, which depend on functions *splitWob* and *orderSub* from Figure 4.7. Prior to starting this stage, a single rewrite to split wobbly relations and sort inequality constraints is necessary, that is the role of *SO*. Afterwards, we proceed with the unification of $[K_i]$, done *UI-base*, and of $[K_w]$, done by *UW-base*. The implementation of this unification round is available in Appendix A.18.

4.2.4 Membership Normalization

In this stage, we assemble the fields that compose a `struct`. To this end, membership relations are first organized. In Figure 4.9, rewrite *SH* sorts *has* constraints in two-levels: (i) by their enclosing α, which identifies a given record; and (ii) by their field name. Placing *has* constraints in such order eases the task of *MN-join*, which creates equivalences between type variables that denote a unique `struct` (rules *MN-skip* and *MN-nfld* are essentially terminators – the latter is necessary for the case where no field membership relations exist). These new equivalences are then unified with $\mathcal{U}_c$, the classical unification algorithm from Figure 4.2. – we refer to this rule as *UE-again*, since its implementation is roughly the same as that of *UE*, from the 1^{st} unification round, in Figure 4.6.

Even though it is not shown in Figure 4.9, the procedure that we have just described must be repeated until *convergence* of field membership relations, as defined by the algorithm in Figure 4.10. By convergence, we mean that, in order to discover that a `struct` field is a pointer to a value whose the type is that same `struct`, it is necessary to unify the enclosing α of all the *has* constraints. Yet, such unification may, in turn, lead to the discovery of further *has* constraints whose field is of the α type in question. This goes on and on... But given that it is impossible to create infinite types in C or μC, rewrite $\overset{\circ}{\rightsquigarrow}$ always terminates. This fact is formally stated by Lemma 7. Appendix A.19 shows the implementation of the membership normalization stage.

Lemma 7 (Termination of Field Membership Convergence). *Let* $\mathcal{C} = \langle \phi, \psi, \Theta, \top, [], [], [], [K_f] \rangle$ *be a configuration. By rewriting* $\mathcal{C}$ *with* $\overset{\circ}{\rightsquigarrow}$*, as defined in Figure 4.10, either the solver halts due to an error, or* $\exists \mathcal{C}_\nu = \langle \phi_\nu, \psi_\nu, \Theta_\nu, \top, [], [], [], [K_f]_\nu \rangle$*, such that* $\mathcal{C} \overset{\circ}{\rightsquigarrow} \mathcal{C}_\nu$.

Proof. First, we show that the rewrites, $\overset{SH}{\rightsquigarrow}$, $\overset{MN}{\rightsquigarrow}*$, and $\overset{UE}{\rightsquigarrow}*$, from Figure 4.9 terminate, or the solver halts due to an error. Second, we show that the condition `if` $[K_f]_{sh} \neq [K_f]_\nu$ fails to hold, eventually.

<u>Case $\overset{SH}{\rightsquigarrow}$</u> *SH.*

sortBy is a standard sorting algorithm, invoked with a *pure* binary predicate, **P**. An error never happens and the function always returns. As a consequence, this rewriting rule terminates.

<u>Case $\overset{MN}{\rightsquigarrow}*$</u> *MN-join.*

While $[K_f]$ is non-empty, with at least two *has* constraints in front of $\perp$, the head of the list is moved to the back, decreasing the distance from the new head to the sentinel.

$\{SH\}$
$\langle \phi, \psi, \Theta, \top, [], [], [], [K_f] \rangle \overset{SH}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [], [], [], [K_f]' \rangle$

```
    where
  $[K_f]' = sortBy(\mathbf{P},\ [K_f])$
  $\mathbf{P}\ (has(\alpha_1, x_1 : \tau_1),\ has(\alpha_2, x_2 : \tau_2)) = $ case $\alpha_1 == \alpha_2 \Longrightarrow x_1 < x_2$
                                                  $\_ \Longrightarrow \alpha_1 < \alpha_2$
```

$\{MN\!-\!join\}$
$\langle \phi, \psi, \Theta, \top, [K_e], [], [], has(\alpha_1, x_1 : \tau_1) :: has(\alpha_2, x_2 : \tau_2) :: [K_f] \rangle \overset{MN}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [K_e]', [], [], [K_f]' \rangle$

```
    where
  $[K_f]' = has(\alpha_2, x_2 : \tau_2) :: [K_f]\ {+}{+}\ [has(\alpha_1, x_1 : \tau_1)]$
  $[K_e]' = $ case $\alpha_1 == \alpha_2\ \&\&\ x_1 == x_2 \Longrightarrow (\tau_1 \equiv \tau_2) :: [K_e]$
                                 $\_ \Longrightarrow [K_e]$
```

$\{MN\!-\!skip\}$
$\langle \phi, \psi, \Theta, \top, [K_e], [], [], has(\alpha, x : \tau) :: \bot :: [K_f] \rangle \overset{MN}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [K_e], [], [], [K_f]' \rangle$

```
    where
  $[K_f]' = [K_f]\ {+}{+}\ [has(\alpha, x : \tau)]$
```

$\{MN\!-\!nfld\}$
$\langle \phi, \psi, \Theta, \top, [], [], [], [\bot] \rangle \overset{MN}{\rightsquigarrow} \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$

$\{UE\!-\!again\}$
$\langle \phi, \psi, \Theta, \top, (\tau_1 \equiv \tau_2) :: [K_e], [], [], [K_f] \rangle \overset{UE}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [K_e]', [], [], [K_f]' \rangle$

```
    where
  $\sigma = \mathcal{U}_c(\tau_1, \tau_2)$
  $\phi' = foreachValue(\sigma, \phi)$
  $\psi' = foreachValue(\sigma, \psi)$
  $\Theta' = foreachValue(\sigma, \Theta)$
  $[K_e]' = \sigma[K_e]$
  $[K_f]' = \sigma[K_f]$
```

Figure 4.9. Rules for the membership normalization stage of the solving process. We omit the definition of function *sortBy*, since it is a standard sorting algorithm. Only the comparison predicate that it uses, **P**, is of interest. Even though *UE-again* is essentially a duplicate of *UE-base*, from Figure 4.6, we keep it in this figure to stress such second use of the classical unification algorithm, $\mathcal{U}_c$ – during this stage, however, both the lists $[K_i]$ and $[K_w]$ are empty. These rewrites are successively invoked by the recursive *converge* function, defined in Figure 4.10. This function ensures that all enclosing type variables inside *has* constraints are unified. A final observation is that $\bot$ is used as a sentinel during processing of $[K_f]$ – this constraint is appended to that list prior to the first invocation. In summary, the idea here is to iterate over *has* relations, producing equivalences among type variables that correspond to `struct` types (note that such constraints are kept in the configuration). Appendix A.16 shows the implementation of this stage.

$$\begin{array}{rll}
\langle \phi, \psi, \Theta, \top, [], [], [], [K_f] \rangle & \overset{\circ}{\rightsquigarrow} & \langle \phi_\nu, \psi_\nu, \Theta_\nu, \top, [], [], [], [K_f]_\nu \rangle \\
 & = & \\
\langle \phi, \psi, \Theta, \top, [], [], [], [K_f] \rangle & \overset{SH}{\rightsquigarrow} & \langle \phi, \psi, \Theta, \top, [], [], [], [K_f]_{sh} \rangle \\
 & \overset{MN}{\rightsquigarrow}* & \langle \phi, \psi, \Theta, \top, [K_e]_{mn}, [], [], [K_f]_{mn} +\!\!+ [\bot] \rangle \\
 & \overset{UE}{\rightsquigarrow}* & \langle \phi_\nu, \psi_\nu, \Theta_\nu, \top, [], [], [], [K_f]_\nu \rangle \\
 & \overset{\circ}{\rightsquigarrow} & \langle \phi', \psi', \Theta', \top, [], [], [], [K_f]' \rangle \quad \texttt{if } [K_f]_{sh} \neq [K_f]_\nu
\end{array}$$

Figure 4.10. The algorithm for convergence of field membership relations. The recursive nature of this rewrite, denoted by $\overset{\circ}{\rightsquigarrow}$, is on par with the existence of recursive types. We may only discover that a field of a given `struct` is a pointer to the type of this same `struct`, if the enclosing α of a *has* constraint containing such field type gets unified. Such discovery triggers further unifications which may eventually lead the discovery of recursive fields at a deeper (more nested) level. Because the definition of infinite types is impossible in C or μC, this rewrite always terminates. The implementation of $\overset{\circ}{\rightsquigarrow}$ is available in Appendix A.20.

Case $\overset{MN}{\rightsquigarrow}*$ *MN-skip.*

While $[K_f]$ is non-empty, with a single *has* constraint in front of $\bot$, the head of this list is moved to the back, and the sentinel is reached.

Case $\overset{MN}{\rightsquigarrow}*$ *MN-nfld.*

This rule terminates immediately. It is only matched when no field membership relations exist, *i.e.*, $[K_f]$ is originally empty.

Case $\overset{UE}{\rightsquigarrow}*$ *UE-again.*

While $[K_e]$ is non-empty, the head of this list is extracted, and τ_1 and τ_2 are passed as arguments of a call to $\mathcal{U}_c$. If unification halts due to an error, then the solver halts as well. Otherwise, we take a step to a configuration with size of $[K_e]$ decreased by one.

Condition $[K_f]_{sh} \neq [K_f]_\nu$.

By induction on the number of new equivalence constraints, $N_\equiv$, created by *MN-join* in $[K_e]$.

– Basis: $N_\equiv = 1$. We have a single new equivalence, $\tau_1 \equiv \tau_2$. Then only two matching membership relations $has(\alpha, x : \tau_1)$ and $has(\alpha, x : \tau_2)$ exist. By *UE-again*, τ_1 and τ_2 are passed as arguments of a call to $\mathcal{U}_c$. Given that unification does not halt due to an error, σ is either the trivial substitution, `[]`, or a unifier `[`$\alpha_x \mapsto \tau$`]`. In the former case, $[K_f]_{sh} == [K_f]_\nu$ holds and $\overset{\circ}{\rightsquigarrow}$ terminates. In the latter, the aforementioned constraints become $has(\alpha, x : \tau)$ and $has(\alpha, x : \tau)$. A subsequent unification between these fields types (occurring in the next iteration) returns the trivial substitution, leading to $[K_f]_{sh} == [K_f]_\nu$, and, consequently, to the termination of $\overset{\circ}{\rightsquigarrow}$.

– Induction: Let us assume that our claim holds for $N_\equiv = \mathcal{N}$, for any $\mathcal{N} > 1$. We show that it continues to hold for $N_\equiv = \mathcal{N}+1$. In such setup, $\mathcal{N}$ matching membership relations $has(\alpha, x : \tau_1)$, $has(\alpha, x : \tau_2)$, $\dots$, $has(\alpha, x : \tau_{\mathcal{N}-1})$, $has(\alpha, x : \tau_{\mathcal{N}})$ exist. By the hypothesis, either the solver halts due to an error, or it terminates when processing relations $has(\alpha, x : \tau_1)$, $has(\alpha, x : \tau_2)$, $\dots$, $has(\alpha, x : \tau_{\mathcal{N}-1})$. In the latter case, we know that a unifier $\sigma = [\alpha_{1-2} \mapsto \tau_{1-2}, \dots, \alpha_x \mapsto \tau]$ exists. At this point, the arrangement is similar to that of the basis: when matching $has(\alpha, x : \tau)$ and $has(\alpha, x : \tau_{\mathcal{N}})$, either we have a trivial substitution and $\overset{\circ}{\rightsquigarrow}$ terminates because $[K_f]_{sh} == [K_f]_\nu$, or another unifier is computed and termination happens in the next iteration.

$\square$

4.2.4.1 Structural Form

Until now, the first four stages of the solving process have been presented: preprocessing, 1^{st} unification round, 2^{nd} unification round, and membership normalization. Once the rules from those groups have been evaluated, we reach a distinguished configuration state where: (i) the only constraints remaining to be processed are the ones in $[K_f]$ - those $has(\alpha, x : \tau)$ relations will contain a type variable as their enclosing type and, possibly, as their field type too; (ii) all existentially quantified type variables, except for those in $[K_f]$, are mapped to ground types in ϕ. This setup is, structure-wise, a solution to our type inference problem! The types mapped by ϕ result from a substitution (list) known as a the *most general unifier*.

Although the structure of records is complete at this point, we still need to combine $has(\alpha, x : \tau)$ constraints and extract `struct` definitions out of them. Since C adopts a nominal type system, the records we create must be made synonyms to the type specifiers appearing in a program. But before entering into the details of this process, let us formalize the current state of our solver: such distinguished configuration is known as a *structural solved form*, as stated by Definition 11. The use of term *structural* is on pair with the idea of a structural type system. Throughout the text, other variations of a solved form will be presented, in particular one that is appropriate to the nominal type system of μC. In Definition 11, the backslash, $\backslash$, denotes the set difference operation. Example 6 illustrates it. Program variables in ψ whose type cannot be determined are the subject of Section 4.2.6.

Definition 11 (Structural Solved Form)**.** Let $\mathcal{C}_i = \langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle$ be an initial configuration implied by a constraint K. A structural solved form of $\mathcal{C}_i$ is a configuration $\mathcal{C}_s = \langle \phi, \psi, \Theta, \top, [], [], [], [K_f] \rangle$ such that:

$$\bigcup_{\{\alpha,\tau\}}^{\phi} \mathit{ftv}(\tau) = \varnothing, \ \ \forall \alpha \in K \setminus (\overline{\alpha_f} \cup \overline{\alpha_\psi} \cup \overline{\alpha_\Theta})$$

$$\text{where:} \ \ \overline{\alpha_f} = \bigcup_{\mathit{has}(\alpha,x:\tau)}^{[K_f]} \alpha, \ \ \overline{\alpha_\psi} = \bigcup_{\{x,\tau\}}^{\psi} \mathit{ftv}(\tau), \ \ \overline{\alpha_\Theta} = \bigcup_{\{\widehat{\tau},\tau\}}^{\Theta} \mathit{ftv}(\tau)$$

◇

Example 6. The structural solved form of the example in Figure 3.11, which presents the constraints produced for the program in Figure 1.3, has $\phi = \{\{\alpha_0, \texttt{int}\}, \{\alpha_1, \texttt{int}*\}, \{\alpha_2, \texttt{int}*\}, \{\alpha_3, \texttt{int}*\}, \{\alpha_4, \texttt{int}\}, \{\alpha_5, \texttt{int}\}, \{\alpha_6, \texttt{int}\}, \{\alpha_7, \texttt{int}\}, \{\alpha_8, \texttt{int}*\}, \{\alpha_9, \texttt{int}\}\}$, $\psi = \{\{c, \texttt{int}*\}, \{f, \texttt{int}(*)()\}\}$, $\Theta = \{\{\text{"T"}, \texttt{int}*\}, \{\text{"int"}, \texttt{int}\}, \{\text{"double"}, \texttt{double}\}\}$, and $[K_f] = []$.

We bring to attention the fact that, different than Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005] and Odersky *et al.* [Odersky et al., 1999], our solved forms, including the structural one from Definition 11 and others that latter appear, are stated by means of a configuration $\mathcal{C}$, and not by means of K. The reason why we do it this way is because, in a nominal type system, it is necessary to decouple the definitions of types from their names, a distinction accomplished by the pairs contained in Θ. In addition, such nominal nature of C and μC prevents us from having a fully solved form where $\mathit{has}(\alpha, x : \tau)$ constraints still exit. Despite this variation in notation, a configuration-based solved form continues to represent a satisfiable constraint.

4.2.5 Record Composition

This group consists of the rules in Figure 4.11. When we arrive at this stage of the solving process, field membership relations will always have a type variable as their *enclosing* type, like in $\mathit{has}(\alpha, x : \tau)$; possibly, they will have a type variable also as their *field type* too, like in $\mathit{has}(\alpha, y : \alpha_f*)$ - this latter situation happens for nested records (including pointer indirections). Our task is to create a new `struct` for each unique α in $[K_f]$. To accomplish that, we traverse *has* constraints and apply either one of the rules *RC-inst* or *RC-upd*. The first rule, *RC-inst*, is responsible for type instantiation. The name of the `struct` that it creates is composed by the typeid of

$\{RC\!-\!inst\}$

$\langle \phi, \psi, \Theta, \top, [], [], [], \; has(\alpha, x : \tau) :: [K_f] \; \rangle \overset{RC}{\rightsquigarrow} \langle \; \phi' \;, \psi, \; \Theta' \;, \top, [], [], [], \; [K_f]' \; \rangle$

`where`

$n = \text{“TYPE\_”} +\!\!+ \; \widehat{\alpha}$

$\tau_s = \texttt{struct} \; n \; \texttt{\{}\tau \; x\texttt{\}}$

$\sigma = \texttt{[}\alpha \mapsto \mathcal{T}_n\texttt{]}$

$\phi' = \mathit{foreachValue}(\sigma, \phi)$

$\Theta' = \mathit{foreachValue}(\sigma, \Theta \cup \{n, \, \tau_s\})$

$[K_f]' = \sigma[K_f]$

$\{RC\!-\!upd\}$

$\langle \phi, \psi, \Theta, \top, [], [], [], \; has(\mathcal{T}_n, x : \tau) :: [K_f] \; \rangle \overset{RC}{\rightsquigarrow} \langle \phi, \psi, \; \Theta(n) \leftarrow \tau_r' \;, \top, [], [], [], [K_f] \rangle$

`where`

$\tau_r = \texttt{struct} \; y \; \texttt{\{}D^\star\texttt{\}} = \Theta(n)$

$\tau_r' = \texttt{if} \; x \in D^\star \; \texttt{then} \; \tau_r \; \texttt{else struct} \; y \; \texttt{\{}(\tau \; x) :: D^\star\texttt{\}}$

Figure 4.11. The record composition rules of the solving process. Because our type system is a nominal one, a mapped type in ϕ either is an uninstantiated α, or a τ whose typeid has an associated record definition in Θ. The meaning of operation $\Theta(n) \leftarrow \tau_r'$ is that the definition of record type τ_r is updated with a new field, making it a record τ_r' (note that $\widehat{\mathcal{T}_n} = n$). We remind the reader that, as defined by Figure 3.1, $D^\star$ stands for the list of fields of a record. The name used to instantiate a `struct` is composed by “TYPE_” plus the typeid of the α in the *has* constraint. A `typedef` to this name is later generated. Appendix A.21 contains the implementation of the record composition stage.

the α in question plus the prefix “TYPE_". The choice of this name is, nevertheless, arbitrary[2], as long as we later declare a `typedef` that maps it to the named type $\mathcal{T}_n$ under which it appears in the program. Example 7 illustrates this mechanism. The second rule, *RC-upd*, processes records that have already been instantiated and update their fields. Appendix A.21 shows the implementation of the record composition stage.

Example 7. Consider program $\mathcal{P}$ = `int f() {T v; v->field=42; return 0;}`. Assume that we reconstruct `T` as a record named `TYPE_23`. In order to make $\mathcal{P}$ a valid program, it is only necessary that declaration `typedef struct TYPE_23 {int field;} T;` is provided.

[2]In practice, one needs to be careful not to introduce a name that collides with a name already existing in the program.

4.2.5.1 Nominal Form

This stage, record composition, combined with the previously presented ones, preprocessing, 1^{st} and 2^{nd} unification rounds, and membership normalization, comprise the core of our solver. By successive application of the rules in Figures 4.5, 4.6, 4.8, 4.11, and 4.9, we are able to reconstruct any missing type declaration in a μC program. We refer to such configuration where records are fully defined and can be identified by their names as a *nominal solved form.* A nominal solved form, formalized by Definition 12, is an elaboration of the *structural solved form* from Definition 11. In order to enhance our solver so that the entire C language is covered, a few extra steps are necessary (*e.g.*, handling of variadic functions and array decaying). Those are briefly discussed in Section 5.

Definition 12 (Nominal Solved Form). Let $\mathcal{C}_i = \langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle$ be an initial configuration implied by a constraint K. A nominal solved form of $\mathcal{C}_i$ is a configuration $\mathcal{C}_n = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$, where $\overline{\alpha_\psi}$ and $\overline{\alpha_\Theta}$ are as in Definition 11, and the following holds:

$$\bigcup_{\{\alpha,\tau\}}^{\phi} \mathit{ftv}(\tau) = \varnothing, \ \ \forall \alpha \in K \setminus (\overline{\alpha_\psi} \cup \overline{\alpha_\Theta})$$

◇

4.2.6 Insufficient Information and De-orphanization

Our approach to deal with orphans is inspired by Haskell 98, which defaults instantiation of such types to the $\star$ Haskell kind [Faxén, 2002, Peyton Jones et al., 2003]. In our solver, defaulting rules are employed as well. But it is necessary that we distinguish among a few different cases, depending on both absent and existing syntax in the program. At this point, we extend the scope of the discussion beyond μC, since there is a relevant aspect of C that must be addressed for orphan types. Our de-orphanization logic is now explained. Example 8 illustrates it. Using the terminology introduced with Figure 3.1, consider x to be a program variable that is neither initialized nor used:

1. If no local declaration for x is available, we instantiated its type as `int`.
2. Otherwise, when a local declaration $\tau\ x$; exists, we proceed in one of two ways:

   2.1. If τ is a named type $\mathcal{T}_n$ prefixed by "struct", τ is instantiated as a record whose name is matched with a type synonym, *i.e.*, a `typedef`.

   2.2. Otherwise, the ($*$ and `const` deconstructed) type is instantiated as `int`.

Example 8. In program `int f() { a; return 0;}`, a declaration `T a;` is implicitly created and the orphan `T` is instantiated as `int`. This is an example of case 1, above. For an example of case 2.1, consider program `int f() { struct T a; return 0;}`. Now, `T` must be instantiated as a record. But in program `int g() { T a; return 0;}`, which reflects case 2.2, we can, again, use `int` to de-orphanize `T`.

The notion of an orphan type is formalized by Definition 13. Essentially, an orphan is a variable that remains uninstantiated in ψ and Θ after we reached a nominal solved form. Toward cleanliness of the presentation, we explained in Section 3.1 that a μC program is not allowed to contain undeclared variables – those would correspond to global variables in C. Thus, the rule in Figure 4.12, where we show the de-orphanization stage, does not handle item 1 in the aforementioned list. Nevertheless, PsycheC, the tool we introduce in Chapter 5, covers all the possible situations.

Definition 13 (Orphans)**.** Let $\mathcal{C}_n = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$ be a configuration in nominal solved form, as according to Definition 12. The set of orphan type variables, *otv*, of $\mathcal{C}_n$ is given by $otv(\mathcal{C}_n) = \overline{\alpha_\psi} \cup \overline{\alpha_\Theta}$, where $\overline{\alpha_\psi}$ and $\overline{\alpha_\Theta}$ are as in Definition 11. $\diamond$

The de-orphanization stage is the closure of our type inference. Once rule *DO* from Figure 4.12 is evaluated, it is guaranteed that every type variable that has been existentially quantified is now instantiated, and that each ground type has its name mapped to its definition in Θ. At this point, our solver configuration is a sound solution to the constraints that were originally produced. Yet, one task still remains to be performed: to rewrite Θ back to the syntax of μC However, we do not further describe this process because it is mostly an engineering task - we encourage the reader, nevertheless, to inspect such implementation in both μC and PsycheC. We refer to the configuration reached at this stage as a *complete solved form.* as stated by Definition 14.

Definition 14 (Complete Solved Form)**.** Let $\mathcal{C}_i = \langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle$ be an initial configuration implied by a constraint K. A complete solved form of $\mathcal{C}_i$ is a configuration $\mathcal{C}_c = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$ such that:

$$\bigcup_{\{\alpha,\tau\}}^{\phi} ftv(\tau) = \varnothing, \ \ \forall \alpha \in K$$

$\diamond$

We now establish certain properties about our constraint solver, defined in Figure 4.13. This complete algorithm is denoted by the rewrite $\overset{*}{\rightsquigarrow}$. First, we claim that it is strongly normalizing. This statement is formalized by Theorem 2. Termination

```
$\{DO\}$
$\langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle \overset{DO}{\rightsquigarrow} \langle \phi', \psi', \Theta', \top, [], [], [], [] \rangle$
   where
  $bind\ ((\widehat{\tau},\ \alpha) :: [\ldots]) =$ if $\widehat{\tau}\ startsWith$ "struct "
                                then [$\alpha \mapsto$ struct $\widehat{\tau}$ { int dummy }] $:: (bind\ [\ldots])$
                                else [$\alpha \mapsto$ int] $:: (bind\ [\ldots])$
  $bind\ (\_ :: [\ldots]) = bind\ [\ldots]$
  $bind\ [] = []$

  $\sigma = bind\ [\{\widehat{\tau}, \tau\} \mid \forall \tau\ \in\ \Theta]$
  $\phi' = foreachValue(\sigma, \phi)$
  $\psi' = foreachValue(\sigma, \psi)$
  $\Theta' = foreachValue(\sigma, \Theta)$
```

Figure 4.12. The de-orphanization rule of the solving process. We create a list of substitutions σ for each pair of elements in Θ. By default, an orphan is instantiated as an `int`. Unless, there is syntax in the program that demands the orphan to be instantiated as a record. In such a case, we create a `struct` with a dummy field (the C standard prohibits a record with a non-empty field list [ISO-Standard, 2011]{§6.7.2.1.8}). Note the difference between brackets [] and `[]`. The former denotes a list, while the latter is part of our notation for a substitution. Therefore, in `[`$\alpha \mapsto$ `int]` : $(bind\ [\ldots])$, we are prepending to a list. *startsWith* is an auxiliary function that tests the prefix of typeid string. The implementation of de-orphanization is available in Appendix A.22.

is easily observable: Our set of rewrites is limited. At every iteration, the algorithm either transforms a constraint or eliminates it, causing a side effect upon ϕ, ψ, or Θ. Eventually, the auxiliary lists $[K_e]$, $[K_i]$, $[K_w]$, and $[K_f]$ are emptied, except when a unification error happens and the solver halts.

Theorem 2 (Termination of Constraint Solving)**.** *Let* $\mathcal{C}^\triangle = \langle \phi^\triangle, \psi^\triangle, \Theta^\triangle, K, [K_e]^\triangle, [K_i]^\triangle, [K_w]^\triangle, [K_f]^\triangle \rangle$ *be any configuration assigned to* K*. By rewriting* $\mathcal{C}^\triangle$ *with* $\overset{*}{\rightsquigarrow}$*, as defined in Figure 4.13, either the solver halts due to an error, or* $\exists \mathcal{C} = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$*, such that* $\mathcal{C}^\triangle \overset{*}{\rightsquigarrow} \mathcal{C}$*.*

Proof. By case analysis on the rewrites $\overset{PP}{\rightsquigarrow}*$, $\overset{UE}{\rightsquigarrow}*$, $\overset{SO}{\rightsquigarrow}$, $\overset{UI}{\rightsquigarrow}*$, $\overset{UW}{\rightsquigarrow}*$, $\overset{\circ}{\rightsquigarrow}$, $\overset{RC}{\rightsquigarrow}$, and $\overset{DO}{\rightsquigarrow}$, from Figures 4.5, 4.6, 4.8, 4.10, 4.11, and 4.12, respectively.

<u>Case</u> $\overset{PP}{\rightsquigarrow}*$ *PP-has*, *PP-eq*, *PP-iq*, *PP-inst*, *PP-def*, *PP-fun*, etc.

By Lemma 3.

<u>Case</u> $\overset{UE}{\rightsquigarrow}*$ *UE-base.*

$$\begin{array}{lll}
\langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle & \overset{\blacktriangleright}{\leadsto} & \langle \phi_f, \psi_f, \Theta_f, \top, [], [], [], [] \rangle \\
& = & \\
\langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle & \overset{PP}{\leadsto}* & \langle \phi_{pp}, \psi_{pp}, \Theta_{pp}, \top, [K_e]_{pp}, [K_i]_{pp}, [], [K_f]_{pp} \rangle \\
& \overset{UE}{\leadsto}* & \langle \phi_{ue}, \psi_{ue}, \Theta_{ue}, \top, [], [K_i]_{ue}, [], [K_f]_{ue} \rangle \\
& \overset{SO}{\leadsto} & \langle \phi_{so}, \psi_{so}, \Theta_{so}, \top, [], [K_i]_{so}, [K_w]_{so}, [K_f]_{so} \rangle \\
& \overset{UI}{\leadsto}* & \langle \phi_{ui}, \psi_{ui}, \Theta_{ui}, \top, [], [], [K_w]_{ui}, [K_f]_{ui} \rangle \\
& \overset{UW}{\leadsto}* & \langle \phi_{uw}, \psi_{uw}, \Theta_{uw}, \top, [], [], [], [K_f]_{uw} \rangle \\
& \overset{\circ}{\leadsto} & \langle \phi_{ue'}, \psi_{ue'}, \Theta_{ue'}, \top, [], [], [], [K_f]_{ue'} \rangle \\
& \overset{RC}{\leadsto}* & \langle \phi_{rc}, \psi_{rc}, \Theta_{rc}, \top, [], [], [], [] \rangle \\
& \overset{DO}{\leadsto} & \langle \phi_f, \psi_f, \Theta_f, \top, [], [], [], [] \rangle
\end{array}$$

Figure 4.13. The complete solver algorithm. The process consists of successive applications of the rewrites from Figures 4.5, 4.6, 4.8, 4.9, 4.11, and 4.12. Those from Figure 4.9 are encapsulated by the field membership converge rewrite, $\overset{\circ}{\leadsto}$, from Figure 4.10. Appendix A.23 shows the implementation of this algorithm, denoted by $\overset{\blacktriangleright}{\leadsto}$.

While $[K_e]$ is non-empty, the head of this list is extracted, and τ_1 and τ_2 are passed as arguments of a call to $\mathcal{U}_c$. If unification halts due to an error, then the solver halts as well. Otherwise, we take a step to a configuration with size of $[K_e]$ decreased by one.

Case $\overset{SO}{\leadsto}$ *SO.*

By Lemma 6, both *orderSub* and *splitWob* terminate. An error never happens.

Case $\overset{UI}{\leadsto}*$ *UI-base.*

Similar to the proof of $\overset{UE}{\leadsto}$, but the auxiliary list whose size decreases is $[K_i]$. In addition, by Lemma 6, both *orderSub* and *splitWob* terminate.

Case $\overset{UW}{\leadsto}*$ *UW-base.*

Similar to the proof of $\overset{UE}{\leadsto}$, but the auxiliary list whose size decreases is $[K_w]$.

Case $\overset{\circ}{\leadsto}$ *SH*, *MN-join*, *MN-skip*, *MN-nfld*, and *UE-again.*

By Lemma 7.

Case $\overset{RC}{\leadsto}*$ *RC-inst.*

While $[K_f]$ is non-empty, the head of this list is extracted, and a `struct` is composed (with a name based on α, and containing field x). Because the substitution σ is manually constructed, an error never happens. We take a step to a configuration where the size of $[K_f]$ decreases by one.

<u>Case</u> $\overset{RC}{\leadsto}*$ *RC-upd.*
Similar to the proof of *RC-inst*, but, in this rule, we update an already instantiated `struct`.

<u>Case</u> $\overset{DO}{\leadsto}$ *DO.*
An error never happens in this rule. Its termination depends on that of *bind*. At each iteration of this function, the size of $[\ldots]$ decreases; eventually we have $[]$ and *bind* returns. □

During the discussion about the preprocessing stage, we introduced the notion of configuration entailment. This property, which is related to the semantics of Figure 3.4, is actually preserved throughout the entire solving process, but with a caveat: Lemma 4 ensures $\mathcal{C} \equiv \mathcal{C}'$, while in Theorem 3, stated below, the entailment is $\mathcal{C} \Vdash \mathcal{C}'$. A corresponding completeness property, however, cannot be established, since the de-orphanization stage "blindly" instantiates type variables.

Lemma 8 (Preservation of Entailment through ϕ, ψ, Θ, and $\mathcal{K}$)**.** *Let* $\mathcal{C} = \langle \phi, \psi, \Theta, \top, [K_e], [K_i], [K_w], [K_f] \rangle$ *and* $\mathcal{C}' = \langle \phi', \psi', \Theta', \top, [K_e]', [K_i]', [K_w]', [K_f]' \rangle$ *be preprocessed configurations. If* $\phi \preceq \phi'$, $\psi \preceq \psi'$, $\Theta \preceq \Theta'$, $\mathcal{K} \succeq \mathcal{K}'$, *and either assertion* $\mathcal{A}_{\equiv}$ *or* $\mathcal{A}_{\leq}$ *(defined below) hold, then* $\mathcal{C} \Vdash \mathcal{C}'$.

$\mathcal{A}_{\equiv}$: $[K_i] = [K_i]'$ *and* $[K_w] = [K_w]'$.

$\mathcal{A}_{\leq}$: $[K_i]' +\!\!+\, [K_w]'$ *respect the relation* $\mathcal{R}_{\leq}$, *as defined in Section 4.1.*

Proof. By induction on the structure of a type, τ, in the rules *KEq* and *KIq*, from Figure 3.4. □

Theorem 3 (Soundness of Constraint Solving)**.** *Let* $\mathcal{C}^{\triangle} = \langle \phi^{\triangle}, \psi^{\triangle}, \Theta^{\triangle}, K, [K_e]^{\triangle}, [K_i]^{\triangle}, [K_w]^{\triangle}, [K_f]^{\triangle} \rangle$ *be any configuration assigned to* K. *If* $\exists \mathcal{C} = \langle \phi, \psi, \Theta, \top, [K_e], [K_i], [K_w], [K_f] \rangle$, *such that* $\mathcal{C}^{\triangle} \overset{\blacktriangleright}{\leadsto} \mathcal{C}$, *then* $\mathcal{C}^{\triangle} \Vdash \mathcal{C}$.

Proof. By case analysis on the rewrites $\overset{PP}{\leadsto}*$, $\overset{UE}{\leadsto}*$, $\overset{SO}{\leadsto}$, $\overset{UI}{\leadsto}*$, $\overset{UW}{\leadsto}*$, $\overset{\circ}{\leadsto}$, $\overset{RC}{\leadsto}$, and $\overset{DO}{\leadsto}$, from Figures 4.5, 4.6, 4.8, 4.10, 4.11, and 4.12, respectively.

<u>Case</u> $\overset{PP}{\leadsto}*$ *PP-has*, *PP-eq*, *PP-iq*, *PP-inst*, *PP-def*, *PP-fun*, etc.
By Lemma 4.

<u>Case</u> $\overset{UE}{\leadsto}*$ *UE-base.*
By Lemma 5, Corollary 2, and Lemma 8, with assertion $\mathcal{A}_{\equiv}$.

<u>Case</u> $\overset{SO}{\leadsto}$ *SO.*
By Lemma 8, with assertion $\mathcal{A}_{\leq}$.

Case $\overset{UI}{\rightsquigarrow}*$ *UI-base.*

By Lemma 5, Corollary 2, and Lemma 8, with assertion $\mathcal{A}_{\leq}$. For the first iteration, the hypothesis is guaranteed by invocations of *splitWob* and *orderSub* in *SO*. In each following iteration, *UI-base*, itself, invokes those two functions.

Case $\overset{UW}{\rightsquigarrow}*$ *UW-base.*

Similar to the proof of *UI-base*.

Case $\overset{\circ}{\rightsquigarrow}$ *SH*, *MN-join*, *MN-skip*, *MN-nfld*, and *UE-again.*

Similar to the proof of *UE-base*.

Case $\overset{RC}{\rightsquigarrow}*$ *RC-inst.*

The name of the instantiated `struct` is created from the typeid, $\widehat{\alpha}$. By either *SSpc* or *PSpc*, from Figure 3.5, we have $\phi \preceq \phi'$ and $\Theta \preceq \Theta'$, so that $\mathcal{C}^{\triangle} \models \mathcal{C}$ holds.

Case $\overset{RC}{\rightsquigarrow}*$ *RC-upd.*

Our semantics does not account for structural decomposition of a record type (fields themselves are unified during the membership normalization stage), $\mathcal{C}^{\triangle} \models \mathcal{C}$ immediately holds.

Case $\overset{DO}{\rightsquigarrow}$ *DO.*

By any of the predicates from Figure 3.5, except for *SVr*, *SVl*, *PVr*, *PVl*, *SVrv*, *SVlv*, *PVrv*, and *PVlv*, , $\mathcal{C}^{\triangle} \models \mathcal{C}$ holds.

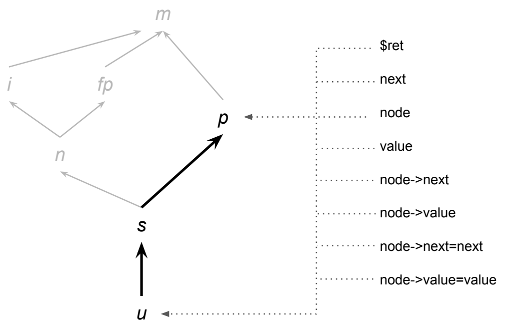
□

Corollary 3 (Complete Instantiability of Constraint Solving). *Given two configurations $\mathcal{C}_i = \langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle$ and $\mathcal{C}_c = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$, if $\mathcal{C}_i \Vdash \mathcal{C}_c$, then $\mathcal{C}_c$ is a complete solved form of $\mathcal{C}_i$.*

Proof. By Theorems 2 and 3. □

To conclude this section, Figure 4.14 shows a complete example of our type inference system, starting from the generation of constraints, until the end of our solving process. The code snippet which we infer types to is based on the function `new_node` from Figure 1.1. In this example, not every rule of our solver is used: wobbly inequalities do not appear and the membership normalization does not exist at all, since there is no multiple uses of a given `struct` field. Even though the deorphanization stage is not illustrated (due to space), type variable α_{12} is an orphan that gets instantiated as `int`.

```
node_t new_node(value_t value,
                node_t next) {
  node_t node;
  node->next = next;
  node->value = value;
  return node;
}
```



$$K = \begin{array}{l} \exists\alpha_0.\ \textit{syn}\ \texttt{node\_t}\ \textit{as}\ \alpha_0 \ \wedge\ \textit{fun}\ \texttt{new\_node:node\_t}\longrightarrow\texttt{value\_t}\longrightarrow\alpha_0 \ \wedge\ \exists\alpha_1.\ \textit{syn}\ \texttt{value\_t}\ \textit{as}\ \alpha_1 \ \wedge\ \textit{def}\ \texttt{value:}\alpha_1\ \textit{in} \\ \exists\alpha_2.\ \textit{syn}\ \texttt{node\_t}\ \textit{as}\ \alpha_2 \ \wedge\ \textit{def}\ \texttt{next:}\alpha_2\ \textit{in}\ \exists\alpha_3.\ \textit{syn}\ \texttt{node\_t}\ \textit{as}\ \alpha_3 \ \wedge\ \textit{def}\ \texttt{node:}\alpha_3\ \textit{in}\ \exists\alpha_4\alpha_5\alpha_6.\ \alpha_6\leq\alpha_5 \ \wedge\ \alpha_5\equiv\alpha_4 \ \wedge\ \exists\alpha_7\alpha_8\alpha_9. \\ \textit{has}(\alpha_8, \texttt{next:}\alpha_9) \ \wedge\ \alpha_7\equiv\alpha_8{*} \ \wedge\ \alpha_9\equiv\alpha_5 \ \wedge\ \textit{typeof}(\texttt{node},\alpha_7) \ \wedge\ \textit{typeof}(\texttt{next},\alpha_6) \ \wedge\ \exists\alpha_{10}\alpha_{11}\alpha_{12}.\ \alpha_{12}\leq\alpha_{11} \ \wedge\ \alpha_{11}\equiv\alpha_{10} \ \wedge\ \exists\alpha_{13}\alpha_{14}\alpha_{15}. \\ \textit{has}(\alpha_{14}, \texttt{value:}\alpha_{15}) \ \wedge\ \alpha_{13}\equiv\alpha_{14}{*} \ \wedge\ \alpha_{15}\equiv\alpha_{11} \ \wedge\ \textit{typeof}(\texttt{node},\alpha_{13}) \ \wedge\ \textit{typeof}(\texttt{value},\alpha_{12}) \ \wedge\ \exists\alpha_{16}.\ \alpha_{16}\leq\alpha_0 \ \wedge\ \textit{typeof}(\texttt{node},\alpha_{16}) \ \wedge\ \top \end{array}$$

Preprocessing

$\Phi = \{\ \{\alpha_0, \alpha_0\}, \{\alpha_1, \alpha_1\}, \{\alpha_2, \alpha_2\}, \{\alpha_3, \alpha_3\}, \{\alpha_4, \alpha_4\}, \{\alpha_5, \alpha_5\}, \{\alpha_6, \alpha_6\}, \{\alpha_7, \alpha_7\}, \{\alpha_8, \alpha_8\}, \{\alpha_9, \alpha_9\}, \{\alpha_{10}, \alpha_{10}\}, \{\alpha_{11}, \alpha_{11}\}, \{\alpha_{12}, \alpha_{12}\}, \{\alpha_{13}, \alpha_{13}\},$
$\{\alpha_{14}, \alpha_{14}\}, \{\alpha_{15}, \alpha_{15}\}, \{\alpha_{16}, \alpha_{16}\},\ \}$

$\psi = \{\ \{\texttt{new\_node}, \alpha_0\texttt{(*) (node\_t, value\_t)}\}, \{\texttt{next}, \alpha_2\}, \{\texttt{node}, \alpha_3\}, \{\texttt{value}, \alpha_1\},\ \}$

$\Theta = \{\ \{\texttt{double}, \texttt{double}\}, \{\texttt{int}, \texttt{int}\}, \{\texttt{node\_t}, \alpha_0\}, \{\texttt{value\_t}, \alpha_1\},\ \}$

$[K_e] = \alpha_3\equiv\alpha_{16}\ \ \alpha_1\equiv\alpha_{12}\ \ \alpha_3\equiv\alpha_{13}\ \ \alpha_{15}\equiv\alpha_{11}\ \ \alpha_{13}\equiv\alpha_{14}{*}\ \ \alpha_{11}\equiv\alpha_{10}\ \ \alpha_2\equiv\alpha_6\ \ \alpha_3\equiv\alpha_7\ \ \alpha_9\equiv\alpha_5\ \ \alpha_7\equiv\alpha_8{*}\ \ \alpha_5\equiv\alpha_4\ \ \alpha_0\equiv\alpha_3\ \ \alpha_0\equiv\alpha_2$

$[K_f] = \textit{has}(\alpha_{14}, \texttt{value:}\alpha_{15})\ \ \textit{has}(\alpha_8, \texttt{next:}\alpha_9)$

$[K_i] = \alpha_{16}\leq\alpha_0\ \ \alpha_{12}\leq\alpha_{11}\ \ \alpha_6\leq\alpha_5$

$[K_w] = []$

1st Unification Round

$\Phi = \{\ \{\alpha_0, \alpha_8{*}\}, \{\alpha_1, \alpha_{12}\}, \{\alpha_2, \alpha_8{*}\}, \{\alpha_3, \alpha_8{*}\}, \{\alpha_4, \alpha_4\}, \{\alpha_5, \alpha_4\}, \{\alpha_6, \alpha_8{*}\}, \{\alpha_7, \alpha_8{*}\}, \{\alpha_8, \alpha_8\}, \{\alpha_9, \alpha_4\}, \{\alpha_{10}, \alpha_{10}\}, \{\alpha_{11}, \alpha_{10}\}, \{\alpha_{12}, \alpha_{12}\},$
$\{\alpha_{13}, \alpha_8{*}\}, \{\alpha_{14}, \alpha_8\}, \{\alpha_{15}, \alpha_{10}\}, \{\alpha_{16}, \alpha_8{*}\},\ \}$

$\psi = \{\ \{\texttt{new\_node}, \alpha_8{*}\texttt{(*) (node\_t, value\_t)}\}, \{\texttt{next}, \alpha_8{*}\}, \{\texttt{node}, \alpha_8{*}\}, \{\texttt{value}, \alpha_{12}\},\ \}$

$\Theta = \{\ \{\texttt{double}, \texttt{double}\}, \{\texttt{int}, \texttt{int}\}, \{\texttt{node\_t}, \alpha_8{*}\}, \{\texttt{value\_t}, \alpha_{12}\},\ \}$

$[K_f] = \textit{has}(\alpha_8, \texttt{value:}\alpha_{10})\ \ \textit{has}(\alpha_8, \texttt{next:}\alpha_4)$

$[K_i] = \alpha_8{*}\leq\alpha_8{*}\ \ \alpha_{12}\leq\alpha_{10}\ \ \alpha_8{*}\leq\alpha_4$

$[K_e] = [K_w] = []$

2nd Unification Round: *unify inequalities*

$\Phi = \{\ \{\alpha_0, \alpha_8{*}\}, \{\alpha_1, \alpha_{12}\}, \{\alpha_2, \alpha_8{*}\}, \{\alpha_3, \alpha_8{*}\}, \{\alpha_4, \alpha_8{*}\}, \{\alpha_5, \alpha_8{*}\}, \{\alpha_6, \alpha_8{*}\}, \{\alpha_7, \alpha_8{*}\}, \{\alpha_8, \alpha_8\}, \{\alpha_9, \alpha_8{*}\}, \{\alpha_{10}, \alpha_{12}\},$
$\{\alpha_{11}, \alpha_{12}\}, \{\alpha_{12}, \alpha_{12}\}, \{\alpha_{13}, \alpha_8{*}\}, \{\alpha_{14}, \alpha_8\}, \{\alpha_{15}, \alpha_{12}\}, \{\alpha_{16}, \alpha_8{*}\},\ \}$

$\psi = \{\ \{\texttt{new\_node}, \alpha_8{*}\texttt{(*) (node\_t, value\_t)}\}, \{\texttt{next}, \alpha_8{*}\}, \{\texttt{node}, \alpha_8{*}\}, \{\texttt{value}, \alpha_{12}\},\ \}$

$\Theta = \{\ \{\texttt{double}, \texttt{double}\}, \{\texttt{int}, \texttt{int}\}, \{\texttt{node\_t}, \alpha_8{*}\}, \{\texttt{value\_t}, \alpha_{12}\},\ \}$

$[K_f] = \textit{has}(\alpha_8, \texttt{value:}\alpha_{12})\ \ \textit{has}(\alpha_8, \texttt{next:}\alpha_8{*})$

$[K_e] = [K_i] = [K_w] = []$

Record Composition

$\Phi = \{\ \{\alpha_0, \texttt{TYPE\_8*}\}, \{\alpha_1, \alpha_{12}\}, \{\alpha_2, \texttt{TYPE\_8*}\}, \{\alpha_3, \texttt{TYPE\_8*}\}, \{\alpha_4, \texttt{TYPE\_8*}\}, \{\alpha_5, \texttt{TYPE\_8*}\}, \{\alpha_6, \texttt{TYPE\_8*}\},$
$\{\alpha_7, \texttt{TYPE\_8*}\}, \{\alpha_8, \texttt{TYPE\_8}\}, \{\alpha_9, \texttt{TYPE\_8*}\}, \{\alpha_{10}, \alpha_{12}\}, \{\alpha_{11}, \alpha_{12}\}, \{\alpha_{12}, \alpha_{12}\}, \{\alpha_{13}, \texttt{TYPE\_8*}\}, \{\alpha_{14}, \texttt{TYPE\_8}\},$
$\{\alpha_{15}, \alpha_{12}\}, \{\alpha_{16}, \texttt{TYPE\_8*}\},\ \}$

$\psi = \{\ \{\texttt{new\_node}, \texttt{TYPE\_8*(*) (node\_t, value\_t)}\}, \{\texttt{next}, \texttt{TYPE\_8*}\}, \{\texttt{node}, \texttt{TYPE\_8*}\}, \{\texttt{value}, \alpha_{12}\},\ \}$

$\Theta = \{\ \{\texttt{TYPE\_8, struct TYPE\_8}\ \{\ \alpha_{12}\ \texttt{value; TYPE\_8* next;}\ \}\ \}, \{\texttt{double}, \texttt{double}\}, \{\texttt{int}, \texttt{int}\},$
$\{\texttt{node\_t}, \texttt{TYPE\_8*}\}, \{\texttt{value\_t}, \alpha_{12}\},\ \}$

$[K_e] = [K_f] = [K_i] = [K_w] = []$

Figure 4.14. A summarized example of our type inference system. On the top-left corner, a function that is based on the code from Figure 1.1. On the top-right corner, the construction of the lattice of shapes. The constraints, K, together with the relevant solver rules, occupy most of the figure. Rules that have no effect on this particular example are not shown, except for the de-orphanization stage, which instantiates α_{12} as `int`.

4.3 Typing a Reduced μC Program

In Section 3.1, we explained that a valid μC program does not necessarily contain definitions for all the types it uses. A program which lacks the declaration of a `struct` or a `typedef` is denominated, by Definition 4, a *reduced* program. In order to type such a program, we need to somehow enrich the typing context. Possibly, with the maps ϕ, ψ, and Θ that we carry in a configuration. We know, from Section 4.2 that we can rewrite $\mathcal{C}_i = \langle \varnothing, \varnothing, \Theta_{std}, K, [], [], [], [] \rangle$ as a complete solved form, $\mathcal{C}_c = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$, as stated by Definition 14. What remains to be done is to verify whether the types in a complete solved form allows us to type check a μC program.

A typical judgment for a *constraint-based* type system would be $\Gamma, K \vdash \mathbf{t} : \tau$, where $\mathbf{t}$ is a typeable term of the language and K is a *satisfiable* constraint. We slightly deviate from Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005] in this regard, since our "satisfiable constraint" is actually a configuration. In particular, one in complete solved form. Our formulation also diverges from that of Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005] in the fact that we do not deal with the generalization and instantiation of *type schemes*. In this aspect, we are closer to the original HM(X) framework by Odersky *et al.* [Odersky et al., 1999].

The typing judgment employed for a reduced μC program is given by $\Gamma, \psi, \Theta \vdash \mathbf{t} : \tau$, where $\mathbf{t}$ can be a function F, a declaration D, an statement S, or an expression E - we do not type the program P as a whole, to make the presentation simpler. The interpretation of this judgment is that $\mathbf{t}$ has type τ within an composite environment consisting of Γ and the mapping information available in ψ and Θ. Therefore, the rules that we previously presented in Figure 3.12 are now enhanced with such larger environment, and shown in Figure 4.15. Most of typing rules remain untouched, except for the few ones which we now discuss. The implementation of our type checking rules appear in Appendix A.24.

Rules *TCDclSt* and *TCPar* are, respectively, the counterparts of rules *TDclSt* and *TPar*. The difference between them is that, in the former, the type of variable x, is obtained from ψ and Θ and included in Γ. Looking at the constraint generation rules from Figure 3.6, this type emerges from *syn* τ *as* α and *def* $x : \alpha$ *in* K. Eventually, upon an use of x, as shown by *TCVar*, τ is fetched from Γ. In this rule, we additionally require that $\Gamma(x) = \psi(x)$. Even though such enforcement is not necessary for the typing itself, it is imposed to ensure correctness of our solving process: for a given declaration τ_d x, the type τ that we make a synonym of τ_d, in Θ, must be the same associated with x, in ψ[3]. A similar check exists in rule *TCRet*.

[3] PsycheC, as opposed to our μC implementation, allows for undeclared variables within a function.

The other typings that have been adjusted from Figure 3.12 are those of rules *TCExpSt*, *TCAsg*, *TCRet*, and *TCFld*. In the first one, *TCExpSt*, we simply require that the type of an expression-statement, albeit discarded, must be a ground one. *TCAsg* and *TCRet* remain essentially the same. The only subtlety is that the subtyping relation now needs to be established on the premises of ϕ, respecting the semantics of Figure 3.5. In regards to *TCFld*, the modification is that we make it explicit the retrieval of a `struct` definition by looking up its name in Θ.

By ensuring that the typings in Figure 4.15 hold, we guarantee soundness of our constraints and thus of our type inference. This property is formally stated, at the expression level, in Theorem 4. Extending this claim to entire functions and to the overall program is straightforward - although the proof would involve no additional concept, extra machinery would be necessary. A property of completeness, however, cannot be established for our type inference system. Besides the incompleteness inherent to the solver of Section 4.2, we do not account for *value categories* as specified in C [ISO-Standard, 2011]{§6.3.2.1}. For instance, although `p = &42;` can be verified from a type checking perspective (assuming `p` is `int*`), this is not a valid statement, since taking the address of a literal is a forbidden operation.

Theorem 4 (Soundness of Constraint Generation - for Expressions). *Let E be an expression of a self-contained program $\mathcal{P}$, and $\mathcal{P}'$ be a reduced program derived from $\mathcal{P}$, whose table of shapes is $\mathcal{M}$. Given a configuration $\mathcal{C}_e = \langle \phi_e, \psi_e, \Theta_e, \langle\langle\, E : \tau, \mathcal{M}\,\rangle\rangle_e, [K_e]_e, [K_i]_e, [K_w]_e, [K_f]_e \rangle$, if $\Gamma \vdash E : \tau$ holds in $\mathcal{P}$, then $\exists \mathcal{C} = \langle \phi, \psi, \Theta, \top, [K_e], [K_i], [K_w], [K_f] \rangle$, such that $\mathcal{C}_e \Vdash \mathcal{C}$, and $\Gamma, \psi, \Theta \vdash E : \tau'$, holds in $\mathcal{P}'$, where $\tau' <: \tau$.*

Proof. By induction on the structure of E in the typing rules from Figure 3.12. We show the proof for cases *TAsg* and *TDiv*. The others follow similar principles.

<u>Case</u> *TAsg*.

We have that $\Gamma \vdash E = E_s : \tau$ is well typed. By *TAsg*, from Figure 3.12, $\Gamma \vdash E : \tau$ and $\Gamma \vdash E_s : \tau_s$, where $\tau_s <: \tau$. We want to prove that $\Gamma, \psi, \Theta \vdash E = E_s : \tau'$, where $\tau' <: \tau$, is also well typed, given the configuration $\mathcal{C}_e = \langle \phi_e, \psi_e, \Theta_e, \langle\langle\, E = E_s : \alpha, \mathcal{M}\,\rangle\rangle_e, [K_e]_e, [K_i]_e, [K_w]_e, [K_f]_e \rangle$. To this end, it is necessary to demonstrate that the following hold: (i) $\Gamma, \psi, \Theta \vdash E : \tau'$; (ii) $\Gamma, \psi, \Theta \vdash E_s : \tau_{s'}$; and (iii) $\tau_{s'} <: \tau'$;

1. By the constraint generation rules from Figure 3.6, we have that $\mathcal{C}_e = \langle \phi_e, \psi_e, \Theta_e, K_\exists.\ K_e \wedge K_{es} \wedge K_{kd} \wedge K_{sel}, [], [], [], [] \rangle$.

When that is the case, x will not be found by *TCVar* within $\Gamma(x)$, but it will exist under $\psi(x)$.

$$\frac{}{\Gamma, \psi, \Theta \vdash \ell : \rho(\ell)}\ \{\mathit{TCLit}\} \qquad \frac{\Gamma(x) = \psi(x) = \tau \quad \mathit{ftv}(\tau) = \varnothing}{\Gamma, \psi, \Theta \vdash x : \tau}\ \{\mathit{TCVar}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E : \tau_s{*} \quad \texttt{field}(x, \Theta(\widehat{\tau}_s)) = \tau}{\Gamma, \psi, \Theta \vdash E\texttt{->}x : \tau}\ \{\mathit{TCFld}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E : \tau_p \quad \tau = \tau_p{*}}{\Gamma, \psi, \Theta \vdash {*}E : \tau}\ \{\mathit{TCDrf}\} \qquad \frac{\Gamma, \psi, \Theta \vdash E : \tau}{\Gamma, \psi, \Theta \vdash \&E : \tau{*}}\ \{\mathit{TCAdr}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E : \tau \quad \Gamma, \psi, \Theta \vdash E_s : \tau_s \quad \tau_s <: \tau}{\Gamma, \psi, \Theta \vdash E = E_s : \tau}\ \{\mathit{TCAsg}\} \qquad \frac{\Gamma, \psi, \Theta \vdash E : \tau \quad \mathit{sc}(\tau)}{\Gamma, \psi, \Theta \vdash E = \texttt{0} : \tau}\ \{\mathit{TCAsgZr}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E_1 : \tau_1 \quad \mathit{sc}(\tau_1) \quad \Gamma, \psi, \Theta \vdash E_2 : \tau_2 \quad \mathit{sc}(\tau_2) \quad \Gamma \vdash || : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{int}}{\Gamma, \psi, \Theta \vdash E_1 \mid\mid E_2 : \tau}\ \{\mathit{TCOr}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E_1 : \tau_1 \ \ \mathit{ari}(\tau_1) \ \ \Gamma, \psi, \Theta \vdash E_2 : \tau_2 \ \ \mathit{ari}(\tau_2) \quad \Gamma \vdash / \ : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{rank}(\tau_1, \tau_2)}{\Gamma, \psi, \Theta \vdash E_1 \ / \ E_2 : \tau}\ \{\mathit{TCDiv}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E_1 : \tau_1{*} \quad \Gamma, \psi, \Theta \vdash E_2 : \texttt{int} \quad \Gamma \vdash +_{ptr-int} : \tau_1{*} \rightarrow \texttt{int} \rightarrow \tau_1{*}}{\Gamma, \psi, \Theta \vdash E_1 + E_2 : \tau}\ \{\mathit{TCAddPtrInt}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E_1 : \texttt{int} \quad \Gamma, \psi, \Theta \vdash E_2 : \tau_2{*} \quad \Gamma \vdash +_{int-ptr} : \texttt{int} \rightarrow \tau_2{*} \rightarrow \tau_2{*}}{\Gamma, \psi, \Theta \vdash E_1 + E_2 : \tau}\ \{\mathit{TCAddIntPtr}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E_1 : \tau_1 \ \ \mathit{ari}(\tau_1) \ \ \Gamma, \psi, \Theta \vdash E_2 : \tau_2 \ \ \mathit{ari}(\tau_2) \quad \Gamma \vdash + : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{rank}(\tau_1, \tau_2)}{\Gamma, \psi, \Theta \vdash E_1 + E_2 : \tau}\ \{\mathit{TCAddAri}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E : \tau \quad \mathit{ftv}(\tau) = \varnothing \quad \Gamma, \psi, \Theta \vdash \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma, \psi, \Theta \vdash \texttt{\{}\,E;\ S^\star\texttt{\}} : \tau_r}\ \{\mathit{TCExpSt}\}$$

$$\frac{(\Gamma, x : \Theta(\widehat{\psi(x)})), \psi, \Theta \vdash \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma, \psi, \Theta \vdash \texttt{\{}\tau\ x;\ S^\star\texttt{\}} : \tau_r}\ \{\mathit{TCDclSt}\} \qquad \frac{\Gamma, \psi, \Theta \vdash \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma, \psi, \Theta \vdash \texttt{()}\ \texttt{\{}S^\star\texttt{\}} : \tau_r}\ \{\mathit{TCBdy}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma, \psi, \Theta \vdash \tau_r\ f\texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r}\ \{\mathit{TCFun}\} \qquad \frac{(\Gamma, x : \Theta(\widehat{\psi(x)})), \psi, \Theta \vdash \texttt{(}D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r}{\Gamma, \psi, \Theta \vdash \texttt{(}\tau\ x,\ D^\star\texttt{)}\ \texttt{\{}S^\star\texttt{\}} : \tau_r}\ \{\mathit{TCPar}\}$$

$$\frac{\Gamma, \psi, \Theta \vdash E : \tau_e \quad \tau_e <: \Theta(\widehat{\tau_r}))}{\Gamma, \psi, \Theta \vdash \texttt{\{}\,\texttt{return}\ E;\texttt{\}} : \tau_r}\ \{\mathit{TCRet}\} \qquad \frac{\Gamma, \psi, \Theta \vdash \texttt{0} : \Theta(\widehat{\tau_r}) \quad \mathit{sc}(\Theta(\widehat{\tau_r}))}{\Gamma, \psi, \Theta \vdash \texttt{\{}\,\texttt{return}\ \texttt{0};\texttt{\}} : \tau_r}\ \{\mathit{TCRetZro}\}$$

Figure 4.15. The type checking rules of a μC program, with type inference. Most of them are similar to those in Figure 3.12. The rules now account for the mappings ϕ, ψ, and Θ, extracted from a configuration in complete solved form $\mathcal{C}_c = \langle \phi, \psi, \Theta, \top, [], [], [], [] \rangle$. Such configuration corresponds to a satisfiable constraint. Appendix A.24 contains the implementation of these typing rules.

2. By *PP-and*, from Figure 4.5, $\mathcal{C}_e$ is eventually decomposed into $\mathcal{C}_{e'} = \langle \phi_e, \psi_e, \Theta_e, K_{e1}, [], [], [], [] \rangle = \langle \phi_e, \psi_e, \Theta_e, \langle\langle\, E : \alpha_1, \mathcal{M}\,\rangle\rangle_e, [], [], [], [] \rangle$, and three other configurations, $\mathcal{C}_{e''}$, $\mathcal{C}_{e'''}$, and $\mathcal{C}_{e''''}$.

3. By the hypothesis, $\exists \mathcal{C}_{e1} = \langle \phi_{e1}, \psi_{e1}, \Theta_{e1}, \top, [K_e]_{e1}, [K_i]_{e1}, [K_w]_{e1}, [K_f]_{e1} \rangle$, such that $\Gamma, \psi_{e1}, \Theta_{e1} \vdash E_1 : \alpha_1$, where $\alpha_1 <: \tau_1$.

4. We now have that $\mathcal{C}_{e''} = \langle \phi_{e1}, \psi_{e1}, \Theta_{e1}, K_{es}, [K_e]_{e1}, [K_i]_{e1}, [K_w]_{e1}, [K_f]_{e1} \rangle = \langle \phi_{e1}, \psi_{e1}, \Theta_{e1}, \langle\langle\, E_s : \alpha_2, \mathcal{M}\,\rangle\rangle_e, [K_e]_{e1}, [K_i]_{e1}, [K_w]_{e1}, [K_f]_{e1} \rangle$.

5. By the hypothesis, again, $\exists \mathcal{C}_{e2} = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, \top, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle$, such that $\Gamma, \psi_{e2}, \Theta_{e2} \vdash E_2 : \alpha_2$, where $\alpha_2 <: \tau_2$.

6. We now have that $\mathcal{C}_{e'''} = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, K_{kd}, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, \alpha_2 \leq \alpha_1, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle$.

7. By Lemma 3, $\exists \mathcal{C}_e$, such that $\mathcal{C}_{e'''} \overset{PP}{\rightsquigarrow}{*}\, \mathcal{C}_{kd}$, since an error never happens in *PP-iq*.

8. We now have that $\mathcal{C}_{e''''} = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, K_{kd}, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, \alpha \leq \alpha_1, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle$.

9. By Lemma 3, $\exists \mathcal{C}_e$, such that $\mathcal{C}_{e'''} \overset{PP}{\rightsquigarrow}{*}\, \mathcal{C}_{pp}$, since an error never happens in *PP-iq*.

10. By Lemma 4, $\mathcal{C}_e \equiv \mathcal{C}_{e1} \equiv \mathcal{C}_{e2} \equiv \mathcal{C}_{kd} \equiv \mathcal{C}_{pp}$.

11. By Theorem 3, $\mathcal{C}_{pp} \models \mathcal{C}$, and the following hold: $\Gamma, \psi, \Theta \vdash E : \alpha_1$, where $\alpha_1 <: \tau$, and $\Gamma, \psi, \Theta \vdash E_s : \alpha_2$, where $\alpha_2 <: \tau_s <: \alpha_1$. These satisfy (i) and (ii).

12. By Corollary 3, $\alpha_1 = \tau'$, so that $\mathit{ftv}(\tau') = \varnothing$, $\alpha_2 = \tau_{s'}$, so that $\mathit{ftv}(\tau_{s'}) = \varnothing$, and $\tau_{s'} <: \tau' <: \tau$, satisfying (iii).

Case *TDiv*.
We have that $\Gamma \vdash E_1 \;/\; E_2 : \tau$ is well typed. By *TDiv*, from Figure 3.12, $\Gamma \vdash E_1 : \tau_1$ and $\Gamma \vdash E_2 : \tau_2$. We want to prove that $\Gamma, \psi, \Theta \vdash E_1 \;/\; E_2 : \tau'$, where $\tau' <: \tau$, is also well typed, given the configuration $\mathcal{C}_e = \langle \phi_e, \psi_e, \Theta_e, \langle\langle\, E_1 \;/\; E_2 : \alpha, \mathcal{M}\,\rangle\rangle_e, [K_e]_e, [K_i]_e, [K_w]_e, [K_f]_e \rangle$. To this end, it is necessary to demonstrate that the following hold: (i) $\Gamma, \psi, \Theta \vdash E_1 : \tau_{1'}$, where $\tau_{1'} <: \tau_1$; (ii) $\Gamma, \psi, \Theta \vdash E_2 : \tau_{2'}$, where $\tau_{2'} <: \tau_2$; (iii) $\mathit{ari}(\tau_{1'})$; (iv) $\mathit{ari}(\tau_{2'})$; and (v) $\Gamma \vdash / \; : \tau_{1'} \to \tau_{2'} \to \mathtt{rank}(\tau_{1'}, \tau_{2'})$.

1. By the constraint generation rules from Figure 3.6, we have that $\mathcal{C}_e = \langle \phi_e, \psi_e, \Theta_e, K_\exists .\ K_{e1} \wedge K_{e2} \wedge K_{kd} \wedge K_{sel}, [], [], [], [] \rangle$. In the case of division, $/$, we can safely omit K_{kd}, since K_{sel} is stricter than K_{kd}. Reconsidering, $\mathcal{C}_e = \langle \phi_e, \psi_e, \Theta_e, K_\exists .\ K_{e1} \wedge K_{e2} \wedge K_{sel}, [], [], [], [] \rangle$.

2. By *PP-and*, from Figure 4.5, $\mathcal{C}_e$ is eventually decomposed into $\mathcal{C}_{e'} = \langle \varnothing, \varnothing, \Theta_{std}, K_{e1}, [], [], [], [] \rangle = \langle \phi_e, \psi_e, \Theta_e, \langle\!\langle\, E_1 : \alpha_1, \mathcal{M} \,\rangle\!\rangle_e, [], [], [], [] \rangle$, and two other configurations, $\mathcal{C}_{e''}$ and $\mathcal{C}_{e'''}$.

3. By the hypothesis, $\exists \mathcal{C}_{e1} = \langle \phi_{e1}, \psi_{e1}, \Theta_{e1}, \top, [K_e]_{e1}, [K_i]_{e1}, [K_w]_{e1}, [K_f]_{e1} \rangle$, such that $\Gamma, \psi_{e1}, \Theta_{e1} \vdash E_1 : \alpha_1$, where $\alpha_1 <: \tau_1$.

4. We now have that $\mathcal{C}_{e''} = \langle \phi_{e1}, \psi_{e1}, \Theta_{e1}, K_{e2}, [K_e]_{e1}, [K_i]_{e1}, [K_w]_{e1}, [K_f]_{e1} \rangle = \langle \phi_{e1}, \psi_{e1}, \Theta_{e1}, \langle\!\langle\, E_2 : \alpha_2, \mathcal{M} \,\rangle\!\rangle_e, [K_e]_{e1}, [K_i]_{e1}, [K_w]_{e1}, [K_f]_{e1} \rangle$.

5. By the hypothesis, again, $\exists \mathcal{C}_{e2} = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, \top, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle$, such that $\Gamma, \psi_{e2}, \Theta_{e2} \vdash E_2 : \alpha_2$, where $\alpha_2 <: \tau_2$.

6. We now have that $\mathcal{C}_{e'''} = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, K_{sel}, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle = \langle \phi_{e2}, \psi_{e2}, \Theta_{e2}, \alpha \leq \texttt{double} \wedge \alpha_1 \leq \texttt{double} \wedge \alpha_2 \leq \texttt{double}, [K_e]_{e2}, [K_i]_{e2}, [K_w]_{e2}, [K_f]_{e2} \rangle$. *Note*: We are not strictly correct here, given that, in Figure 3.10, there are multiple cases for the division operator, /. Yet, the case we use for this demonstration is the less strict one.

7. By Lemma 3, $\exists \mathcal{C}_{pp}$, such that $\mathcal{C}_{e'''} \overset{PP}{\rightsquigarrow} * \mathcal{C}_{pp}$, since an error never happens in *PP-iq*.

8. By Lemma 4, $\mathcal{C}_e \equiv \mathcal{C}_{e1} \equiv \mathcal{C}_{e2} \equiv \mathcal{C}_{pp}$.

9. By Theorem 3, $\mathcal{C}_{pp} \models \mathcal{C}$, and the following hold: $\Gamma, \psi, \Theta \vdash E_1 : \alpha_1$, where $\alpha_1 <: \texttt{double} <: \tau_1$, and $\Gamma, \psi, \Theta \vdash E_2 : \alpha_2$, where $\alpha_2 <: \texttt{double} <: \tau_2$. These satisfy (i) and (ii).

10. By Corollary 3, $\alpha_1 = \tau_{1'}$, so that $ftv(\tau_{1'}) = \varnothing$, and $\alpha_2 = \tau_{2'}$, so that $ftv(\tau_{2'}) = \varnothing$. As a consequence, $ari(\tau_{1'})$ and $ari(\tau_{2'})$ hold, satisfying (iii) and (iv).

11. Given (iii) and (iv), satisfaction of (v) is straightforward.

$\square$

Chapter 5

An Overview of PsycheC

PsycheC consists of two components. Constraint generation, as described in Section 3, is implemented in C++ through an AST visitor. Our parser is a modified and extended version of the parser from the `Qt Creator` IDE [Qt-Project, 2017]. The constraint solver is implemented in Haskell. It follows the principles discussed in Section 4.2.

5.1 The C Language

The first standardized version of C, published by ANSI, is known as C89 [ANSI-Standard, 1989][1]. 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 of static analysis tools, the behaviour of compilers, and 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, Runtime-Verification, 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

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

are free (and open source) 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. In its current form, PsycheC addresses the fundamental typing relations of C that are necessary for type inference. As shown in Section 5.3, 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.

5.1.1 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 as described in Section 5.3, less than ten unique syntactically-invalid macros were found.

Example 9. 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 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.

5.1.2 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 `#define`s 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[2], we parse the call but constraints are not generated for arguments.

5.1.3 Arrays x pointers, functions x function pointers

Our constraint's language does not distinguish array access from pointer expressions[3]. 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.

5.1.4 Miscellanea

Below, we list a few other characteristics of PsycheC's implementation.

[2]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.

[3]Array declarations are recognized, but in certain cases it is necessary to analyse further syntax. For instance, a bracket-less declaration like `T var`; could actually hide an array in the case a `typedef int T`[2]; would exist.

– We do not introduce *storage-class specifiers* [ISO-Standard, 2011]{§6.7.1} into declarations, except for a `typedef`, of course. Those do not influence typing.

– By default, field accesses are unified as composing an `struct` – no special recognition for bit-fields [ISO-Standard, 2011]{§6.7.2.1} is employed. A declaration of an `union` is produced by PsycheC only if such a variable appears in the partial program through an elaborated-type-specifier, *e.g.* `union U v;`. Union types are a frequent cause of undefined behavior since they permit *writing* to one field, but *reading* from another.

– C scoping rules allows for an enumerator to be used without qualification. Therefore, it is not possible for PsycheC to match an enumerator to a given enumeration, unless its declaration is in the program on which type inference is performed. Like we do it for `union`s, the declaration of an enumeration is only produced if a variable appears through an elaborated-type-specifier, *e.g.* `enum E e;` (a placeholder is used as an enumerator).

– When PsycheC encounters a name in a context where a *constant-expression* [ISO-Standard, 2011]{§6.6} is required, such as in a `case` statement, a `#define` will be generated. A *constexpr* constraint exists for this purpose.

5.2 A Glimpse of Dynamic Semantics

Despite a varying degree of language-completeness coverage, there has been numerous studies on the formalization of dynamic semantics for C [Ellison and Rosu, 2012, Papaspyrou, 1998, Papaspyrou, 2001, Krebbers, 2015, Krebbers and Wiedijk, 2015, Blazy and Leroy, 2009] and even focused on its concurrency model [Batty et al., 2016, Nienhuis et al., 2016]. Within this matter, we call attention to the fact that, while our type inference produces well typed programs, this does not eliminate the risk of *undefined behaviours*, a topic thoroughly studied by Hathhorn *et al.* [Hathhorn et al., 2015].

PsycheC cannot guarantee that a reconstructed program will behave well at runtime; not even when a partial program is derived from another that is originally free from undefined behaviors. An obvious obstacle that prevents us from establishing stronger dynamic semantics properties is that values may be absent in an incomplete source. In addition to that, there are aspects which our type inference does not account for: (i) inaccuracies resulting from the inference of arithmetic types, which can lead to signed integer overflows; (ii) fields of a composed `struct` may be in different order from those in the original `struct` – just as problematic, what we infer as a `struct` could actually be a `union`; (iii) missing array declarations are always inferred as pointers, possibly leading to unbound memory accesses; (iv) the definition of functions are not synthesized, only their declarations - we note that stub-generation tools [Cadar et al., 2008, Godefroid et al., 2005] can mitigate this problem.

A further observation to make is that PsycheC's reconstruction is cast-free: we do not introduce type casts in the program on which type inference is performed; hence, we do not change type relations already in place. In particular, we respect the contracts [Wadler and Findler, 2009] between types and subtypes that we create to handle qualifiers. Finally, we once more emphasize that our results pertain to the static semantics of programs; hence, undefined behavior does not compromise our ability to discover typing information.

5.3 Empirical Evaluation

The key contribution of this thesis is a technique to infer types in a C program. 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 would one go over all this trouble to reconstruct a C program? The answer to this question is another contribution of our work. In the upcoming sections, we describe practical uses of a type inference engine for incomplete C sources.

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[4]. 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 that require a fully compilable C program.

While we have, throughout our experiments, collected code from numerous open source repositories (and even a few proprietary ones), it is impossible to claim that PsycheC is free from errors or unbiased toward certain programming style. In particular, there is likely some exotic C feature that we have not yet seen, what can be considered a threat to the validity of the evaluation. Nevertheless, we expect that, in regards to typing aspects of C, our implementation is complete.

[4]Earlier in the year of 2017, PsycheC appeared among GitHub's most trending C projects.

5.3.1 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[5], 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 5.1 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 Table 2.1. The parsing technique of Section 2 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 2 is relevant to allow a formal end-to-end approach of our type inference. Table 5.1 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[6] because this compiler does

[5]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`.

[6]Our kcc compilations have been performed through RV-Match, available at: `https://runtimeverification.com/match/`.

Table 5.1. Reconstruction of the GNU Coreutils programs. **LoC:** lines of code in the **O**riginal program and in the **P**artial one; **Disamb:** syntactical ambiguities resolved **A**lgorithmically or **H**euristically; and **S**emantic 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	
	O	P	A	H	S	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

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 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 `switch`s 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.

5.3.2 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` [Larochelle et al., 2001, Evans, 1996], `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`[7], 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 5.1 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 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

[7]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;
```

Figure 5.1. 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.

to the snippets from Figure 5.1, PsycheC produces declarations that lead to the same diagnostics.

5.3.3 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.

Benchmark: `Cppcheck`[8], 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.

5.3.4 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[9]

Discussion: The function displayed in Figure 5.2 has been submitted as a patch[10] 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

[8]Available at `http://cppcheck.sourceforge.net/` on July 2017
[9]Available at `http://pathcrawler-online.com:8080/` on July 2017.
[10]Available at https://github.com/git/git/commit/d70e9c5c8c865626b6e69c2bf9fd0e368543617b

```
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; };
```

Figure 5.2. 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.

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.

5.3.5 Extracting Data-Structures

Goal: Extract complete definitions of data structures from 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* [GNOME-Project, 2017], the

GNU Portability Library *Gnulib* [Free-Software-Foundation, 2017] and the Generic Data Structure Library (*GDSL*) [GDSL-Team, 2017]. In addition, we considered functions from Sedgewick's book [Sedgewick, 2002][11]. 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 5.2 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 5.2 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 varies 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

[11]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 5.2. 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 differentiate between a pointer and an integer; only **P**artially, such as a pointer whose underlying type is unknown; 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

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

Chapter 6

Related Work

The technique that we have presented in this thesis follows from a long string of research in programming languages. In this section, we highlight what we consider to be the prominent contributions that helped us devise what is, to the best of our knowledge, the first type inference mechanism that is capable of dealing with the type system of the C programming language.

6.1 Parsing of Incomplete Sources

There exists a body of work about parsing C and/or C++ in face of missing program parts [Bischofberger, 1993, Koppler, 1997, Knapen et al., 1999, Padioleau, 2009, Moonen, 2001, Gazzillo and Grimm, 2012, McPeak and Necula, 2004]. These previous techniques approach the problem of parsing incomplete programs from a software engineering perspective, and have been used in the construction of tools such as bug finders, code browsers, syntax highlighters and code auto-completers. Parsing of partially available code allows such tools to deal with programs that either contain syntactic errors or that make use of non-standard language extensions. Such shortcomings emerge, in particular, when programs are being edited.

The approaches adopted to parse incomplete code vary depending on their goals. For instance, Padionelau [Padioleau, 2009] combines ad-hoc techniques and heuristics based on typical idioms and conventions used by programmers to build a parser; Moonen2001 [Moonen, 2001] proposes the notion of an *island grammar*, which is a grammar slightly more liberal than the official one for the language in question, yet allowing late refinements; Koppler97 [Koppler, 1997] introduces a so-called *fuzzy parser*, a parser tolerant of programming faults and source incompleteness. However, none of the aforementioned solutions is suitable for type inference, given that the AST produced by the parser may not be completely correct.

To build a precise AST for an incomplete program, a parser must carry the ambiguities that it discovers, until they can eventually be disambiguated - if disambiguation is possible. We can look at this problem from the perspective of *Generalized LR* (GLR) parsers. A GLR is an LR parser that forks itself upon an ambiguous construct and expects that only a single thread of execution will continue until the end of the process. One disadvantage of GLR parsers is their typical performance overhead –an aspect that has been addressed by McPeak and Necula [McPeak and Necula, 2004]. However, McPeak and Necula do not deal with the possible absence of a symbol table.

Along the lines of GLR parsing, Gazillo and Grimm [Gazzillo and Grimm, 2012] propose a technique that is capable of addressing incompleteness in the source code by means of a predefined *configuration* for the preprocessor. The work by Gazillo and Grimm is particularly focused on the interactions between the C grammar and that of the preprocessor. Their approach also accounts for predefined configurations at the AST level; hence, distinguishing the different code paths that come from preprocessing conditionals. It is likely that the parser from Gazillo and Grimm can be adapted to our needs, but it would require that we design a custom configuration for it.

Our parsing strategy can best be compared to the work of Knapen *et al.* [Knapen et al., 1999]. The overall principle for AST disambiguation that we employ is based on ideas introduced by them. Nevertheless, because Knapen *et al.* do not present the operational details of their approach, it is difficult to state precisely how it differs from ours. Furthermore, their work is motivated by software engineering tooling: they do not attempt to *reconstruct* a program (for subsequent type inference) as we do. We emphasize that Knapen *et al.*'s presentation is based in plain English; thus, the formalism that we provide in section 2 is a contribution of our work.

6.2 Type inference

The subject of type inference was independently approached in a number of previous works. Hindley, Milner, and Damas are pioneers in this field [Hindley, 1969, Milner, 1978, Damas and Milner, 1982]; yet, early ideas by Kaplan and Ullman [Kaplan and Ullman, 1978] and Borning and Ingalls [Borning and Ingalls, 1982], on the front of Smalltalk, are worth mentioning as well. Wand [Wand, 1987b] was the first to prove correctness of classical the algorithm. Since then, numerous variations and extensions of type inference mechanisms have been presented. For instance, Wand [Wand, 1987a, Wand, 1988] follows up on the work of Cardelli [Cardelli, 1984] to integrate record types plus inheritance into a type system. As another example, Palsberg and Schwartzbach [Palsberg and Schwartzbach, 1991] further developed type inference in the context of object-oriented programming.

Within a traditional type inference algorithm, one would follow a syntax-directed approach where traversal of the program's abstract syntax tree is interleaved with on-the-fly application of type substitutions. While it is feasible to have an efficient implementation of such technique, this formulation has at least two disadvantages: substantial type manipulation is to be expected, and the process of verifying whether a program is well typed is needlessly tied to an specific process of determining the actual types that validate such statement.

To decouple these two tasks (*i.e.* checking for program well typedness and discovering satisfying types), alternative formulations of type inference have been proposed. Important in this front was the idea of reducing a type inference problem into two parts: (i) an algebra responsible for capturing the restrictions imposed by the program on a type, and (ii) a system for solving the pertinent equations arriving from such modelling [Rémy, 1992]. Eventually, such decomposition strategy became informally known as *constraint generation* and *constraint solving*. Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005] have done significant development on this front by extending the constraint-based HM(X) framework from Odersky *et al.* [Odersky et al., 1999]. Nevertheless, the first explicit association between the properties of a type and the mathematical terminology of a *constraint* is likely due to Cardelli [Cardelli and Wegner, 1985].

Our type inference is implemented after the HM(X) algorithm, but we follow more closely the formulation presented by Pottier and Rémy [Pottier and Rémy, 2003, Pottier and Rémy, 2005], even though we do not deal with type schemes. The main difference between our approach and the one from languages such as Haskell and ML is the fact that we do not have the definition of algebraic types – on the contrary, we are trying to build them. Moreover, we use type promotions, conversions, and restrictions involving expressions in C as a way around parametric polymorphism. Such strategy of leveraging properties of a language type system as an aid to reconstruct well typed programs from partial ones is also employed by Dagenais and Hendren [Dagenais and Hendren, 2008], but in the context of Java.

6.3 Unification and Subtyping

Despite differences in the existing approaches to type inference, the underlying principle by which most of them ensure equivalence between two types remains the same: through the unification algorithm by Robinson [Robinson, 1965], and improvements over it, such as the work by Martelli and Montanari [Martelli and Montanari, 1982]. While unification suits well typical statically typed functional languages, this algorithm, in its classical structure, is not a viable choice for systems that feature subtyping. Such mismatch is inherent to the nature of unification, given that a subtyping relation imposes the need of "unifying" types that are not considered equivalent.

There has been a significant amount of work that mixes unification and subtyping in interesting ways. But prior to entering into a discussion about this topic, we need to settle some terminology. When talking about subtyping it is appropriate to elaborate on the exact form of subtyping under discussion. In the simplest setup, we can view types as belonging to a finite *partially ordered set* (poset): *e.g.*, `int` $<:$ `double`. This interpretation of subtyping requires no notion of type constructors. Mitchel [Mitchell, 1991] presents an strategy to combine unification with such *atomic* relations. Tiuryn and Wand have built on this latter work in order to address recursive types [Tiuryn and Wand, 1993].

In our presentation, we deal with a form of subtyping that cannot be modeled through atomic relations only. First, because it is not possible to propagate a (valid) ordering like `int` $<:$ `double` uniformly through pointers, *i.e.*, `int`$*$ $<:$ `double`$*$ does not type check. Second, because not all ground types can be determined in advance, yet we still need to ensure that relations like $\tau*$ $<:$ `const` $\tau*$ hold. Therefore, our system must account for the influence of type constructors and the structure of constructed types. *Structural* subtyping is the terminology employed in the literature to denote the form of subtyping that appears under such scenario.

Numerous formulations that combine unification with structural subtyping have been presented [Kaes, 1992, Smith, 1994, Fuh and Mishra, 1988, Pottier, 1996, Pottier, 1998, Simonet, 2003]. Despite the varying approaches in each of the techniques, they all share to certain extent a common aspect in regards to the handling of inequality constraints: solving of the subtyping relations is not tightly integrated to the unification algorithm, but left as a separate (*e.g.* as a post-processing) procedure. The usual reason for this separation is to avoid dealing with complex constraint relations that emerge during type inference. In fact, a few of the aforementioned works make a noteworthy effort on proposing constraint simplification methods.

In a recent work, Dolan and Mycroft [Dolan and Mycroft, 2017] presented a technique that deals with subtyping directly as part of the unification problem. A prominent characteristic of their technique is the distinction between *input* and *output* types that participate in a subtyping relation. Based on this premise, they introduce the so-called *biunification*, plus the consequential notion of *bisubstitutions*. In spite of its elegance, this approach also brings non-negligible complexity. Much of this complexity is due to the algebraic manipulations that are necessary to implement lattice operators. This lattice ensures that substitutions are applied in a way that respects the unidirectionality of assignments between type and subtype.

As an attempt to position our work, we emphasize its simplicity and applicability. By simplicity, we mean that the underlying techniques comprising the building blocks of our type inference are well understood and easy to implement. Yet, our approach allows a transparent handling of both type equivalence and type inequality constraints. Precisely, we are capable of dealing with structural subtyping whenever the finite set of structure-modifying type constructors is known in advance (in our case, the $*$ used in pointers). The price of the simplicity of our model is the higher computational complexity of its algorithmic implementation –a consequence of the inequality ordering stage that happens during unification. Nevertheless, such cost is only paid for subtyping relations actually produced. These relations appear in small number, compared to the equivalence constraints used in typical unification, since we take advantage of program constructs that allows us to instantiate type in an exact way.

6.4 The Type System of C and Semantics

There is a significant amount of research that investigate dynamic properties of C [Ellison and Rosu, 2012, Papaspyrou, 1998, Papaspyrou, 2001, Krebbers, 2015, Krebbers and Wiedijk, 2015, Blazy and Leroy, 2009]. However, we are mostly interested on the static semantics of this language, given that our works is about type inference. Nevertheless, we briefly mention aspects related to dynamic semantics because, in certain cases, they have influenced our design.

C's type system has been studied in the context of program analysis research [Steensgaard, 1996, Necula et al., 2002a]. Even though such works apply limited forms of type inference, they do so as a way towards a specific goal. Their formulation is not thorough enough for the task of inferring declarations and type definitions in a program. Previous work focusing on the type system of C and its type checker exists as well: *e.g.*, Smith and Volpano [Smith and Volpano, 1996] propose ML-style polymorphism to the language; Necula *et al.* [Necula et al., 2002b] introduce CCured, a type system for C that offers better safety guarantees to pointer usage (with static verification and dynamic checks); Chandra and Reps [Chandra and Reps, 1999] present, in face of common pointer-related errors, an advanced type checking mechanism based on the memory layout of objects. However, these works modify C; They do not address the exact task of type inference for the language as it is currently standardized.

Recently, Noonan *et al.* [Noonan et al., 2016] presented a type inference algorithm for machine code that correlates with our work. Their tool, *Retypd*, consists of two components: 1) a sound inference algorithm constructed over a type-system that is 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 types. Although there is, at first sight, an apparent overlapping of techniques between theirs and our work, we are dealing with different problems: *Retypd* starts with a program in binary format, extracts types out of it, and reports (translated) C types that could have existed in the source code that originated the program in question. We, on the other hand, must ensure that the types we instantiate exactly match a given source code. There is as well a work by Mycroft [Mycroft, 1999] for the decompilation of target machine code back into C programs. While Mycroft seems to make use of type inference, there is not much material about this topic in his presentation.

Foster *et al.* [Foster et al., 1999] presents a technique for the inference of the qualifiers of a type based on atomic subtyping. They deal with the same problem of modelling the behavior of `const` and how it interacts with pointers. The difference is that, as opposed to the scenario where our type inference works, the formulation of Foster *et al.* assumes that the *base* types (*i.e.*, those "behind" a `const` qualifier or pointer modifier) are known. In other words, they do not need to be discovered. Foster *et al.* models a type qualifier by "lifting" the typing rules so that they continue to hold as they would for the non-qualified types. While this approach works for the inference of qualifiers, it is not sufficient for inference of complete types. With only atomic subtyping, it would not be possible to correctly type relations like `int` $<:$ `double` or `int`$*$ $<:$ `const int`$*$ and `double`$*$ $<:$ `const double`$*$, and, at the same time, report errors for relations such as `int`$*$ $<:$ `double`$*$ and `double`$*$ $<:$ `int`$*$.

Chapter 7

Final Thoughts

This thesis has presented a type inference mechanism for the C programming language. As an application of our technique, we have investigated the problem of understanding programs whose source code may lack declarations (either of variables or those that define types). Throughout the text, we showed (i) how to understand syntactic constructs that are locally ambiguous but which can be disambiguated within a larger context; (ii) a "pre inference" approach that lets us give meaning to a program according to a valid (static) semantics of C, even though symbolic information might not be available for use; and (iii) a constraint generation and constraint solving style of type inference that is capable of satisfying the idiosyncrasies of C's type system.

In this work we also showed a variety of scenarios in which our technique can be employed for the reconstruction of partial C programs, so that static analysis tools (and others that rely on a compilable source code) can still work on them. We believe that the theoretical framework developed by us, and the tool constructed to sustain our ideas, PsycheC, opens up many possibilities for researchers and practitioners.

7.1 Future Work

The existence of a type inference engine for C unveils very interesting possibilities of research concerning the dynamic semantics of that language. Currently, we do not associate size information with array types. Instead, they are made plain pointers like `int*`. Yet, current state-of-the-art symbolic range analyses, 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.

Another question that our ideas bring forward concerns signed integer overflows. Can we leverage the strength of static analyses [Rodrigues et al., 2013] to determine the exact type of scalars that are used in signed arithmetic operations/conversions, and, with that information, offer more precise overflow related diagnostics?

Finally, while writing code with the support of PsycheC, programmers have at their disposal an new programming language: one that reuses C's syntax, but that supports type inference. The impact of this "new language" onto the productivity of C programmers is worth to be assessed in a holistic manner.

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

Publications

A paper containing the essence of the work presented in this thesis has been published in the 2018 edition of *Principles of Programming Languages* (POPL). A preliminary discussion of type inference for a simplified C-like language appears in the 2016 edition of *Simpósio Brasileiro de Linguagens de Programação* (SBLP). While a student in the Computer Science department of the Universidade Federal de Minas Gerais, I was also involved in other projects. Below is a list of publications in which I have participated.

- *Inference of static semantics for incomplete C programs*; **Leandro T. C. Melo**, Rodrigo G. Ribeiro, Marcus R. de Araújo, Fernando M. Q. Pereira; POPL 2018.
- *Inferência de Tipos Dependentes em C*; Marcus R. de Araújo, **Leandro T. C. Melo**, Fernando M. Q. Pereira; SBLP 2017.
- *Compilação Parcial de Programas Escritos em C*; Rodrigo G. Ribeiro, **Leandro T. C. Melo**, Marcus R. de Araújo, Fernando M. Q. Pereira; SBLP 2016.
- *SMOV: Array Bound-Check and Access in a Single Instruction*; Antônio L. M. Neto; **Leandro T. C. Melo**; Omar P. Vilela, Fernando M. Q. Pereira, Leandro B. Oliveira; IEEE ICNS 2016.
- *Protecting Programs Against Memory Violation In Hardware*; Antônio L. M. Neto; **Leandro T. C. Melo**; Omar P. Vilela, Fernando M. Q. Pereira; Leandro B. Oliveira; IEEE Latin America Transactions 2015.
- *NomadiKey: User Authentication for Smart Devices based on Nomadic Keys*; Leonardo Cotta, Artur L. Fernandes, **Leandro T. C. Melo**, Luis F. Saggioro, Frederico Sampaio, Antônio L. M. Neto, Antônio F. Loureiro, Italo Cunha, Leonardo B. Oliveira; IEEE ICC 2016.
- *Teclanômade: Uma solução de autenticação para usuários de dispositivos inteligentes baseada em Teclados Nômades*; Antônio L. M. Neto, Artur L. Fernandes, Frederico Sampaio, Leonardo Cotta, **Leandro T. C. Melo**, Luis F. Saggioro, Leonardo B. Oliveira; SBSeg 2015.

Bibliography

[ANSI-Standard, 1989] ANSI-Standard (1989). ANSI X3.159-1989 - The C programming language.

[Batty et al., 2016] Batty, M., Donaldson, A. F., and Wickerson, J. (2016). Overhauling sc atomics in c11 and opencl. In *POPL*, volume 51, pages 634--648. ACM.

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

[Blazy and Leroy, 2009] Blazy, S. and Leroy, X. (2009). Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263--288.

[Borning and Ingalls, 1982] Borning, A. H. and Ingalls, D. H. H. (1982). A type declaration and inference system for smalltalk. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 133--141, New York, NY, USA. ACM.

[Cadar et al., 2008] Cadar, C., Dunbar, D., and Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209--224. USENIX.

[Cardelli, 1984] Cardelli, L. (1984). A semantics of multiple inheritance. In *Semantics of data types*, pages 51--67. Springer.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471--523.

[Chandra and Reps, 1999] Chandra, S. and Reps, T. (1999). Physical type checking for c. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 66--75. ACM.

[Chugh et al., 2009] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. (2009). Staged information flow for javascript. In *PLDI*, pages 50--62. ACM.

[Cuoq et al., 2012] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-c. In *International Conference on Software Engineering and Formal Methods*, pages 233--247. Springer.

[Dagenais and Hendren, 2008] Dagenais, B. and Hendren, L. (2008). Enabling static analysis for partial java programs. In *OOPSLA*, pages 313--328. ACM.

[Damas and Milner, 1982] Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207--212. ACM.

[Dolan and Mycroft, 2017] Dolan, S. and Mycroft, A. (2017). Polymorphism, subtyping, and type inference in mlsub. In *POPL*, pages 1--13. ACM.

[Dubois and Menissier-Morain, 1999] Dubois, C. and Menissier-Morain, V. (1999). Certification of a type inference tool for ml: Damas–milner within coq. *Journal of Automated Reasoning*, 23(3):319--346.

[Ellison and Rosu, 2012] Ellison, C. and Rosu, G. (2012). An executable formal semantics of c with applications. In *POPL*, volume 47, pages 533--544. ACM.

[Evans, 1996] Evans, D. (1996). Static detection of dynamic memory errors. In *PLDI*, volume 31, pages 44--53. ACM.

[Faxén, 2002] Faxén, K.-F. (2002). A static semantics for haskell. *J. Funct. Program.*, 12(5):295--357.

[Foster et al., 1999] Foster, J. S., Fähndrich, M., and Aiken, A. (1999). A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192--203.

[Free-Software-Foundation, 2017] Free-Software-Foundation (2017). Gnulib - the gnu portability library. `https://www.gnu.org/software/gnulib/`.

[Fuh and Mishra, 1988] Fuh, Y.-C. and Mishra, P. (1988). Type inference with subtypes. In *European Symposium on Programming*, pages 94--114. Springer.

[Gazzillo and Grimm, 2012] Gazzillo, P. and Grimm, R. (2012). Superc: parsing all of c by taming the preprocessor. *ACM SIGPLAN Notices*, 47(6):323--334.

[GDSL-Team, 2017] GDSL-Team (2017). The generic data structures library. `http://home.gna.org/gdsl/`.

[GNOME-Project, 2017] GNOME-Project (2017). The gnome library - glib. `https://developer.gnome.org/glib`.

[Godefroid, 2014] Godefroid, P. (2014). Micro execution. In *ICSE*, pages 539--549. ACM.

[Godefroid et al., 2005] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. In *PLDI*, pages 213--223. ACM.

[Hathhorn et al., 2015] Hathhorn, C., Ellison, C., and Rosu, G. (2015). Defining the undefinedness of C. In *PLDI*, pages 336--345. ACM.

[Hindley, 1969] Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29--60.

[ISO-Standard, 1990] ISO-Standard (1990). ISO/IEC 9899:1990 - The C programming language.

[ISO-Standard, 1999] ISO-Standard (1999). ISO/IEC 9899:1999 - The C programming language.

[ISO-Standard, 2011] ISO-Standard (2011). ISO/IEC 9899:2011 - The C programming language.

[Jones et al., 2004] Jones, S. P., Washburn, G., and Weirich, S. (2004). Wobbly types: type inference for generalised algebraic data types. Technical report, Technical Report MS-CIS-05-26, Univ. of Pennsylvania.

[Kaes, 1992] Kaes, S. (1992). Type inference in the presence of overloading, subtyping and recursive types. In *ACM SIGPLAN Lisp Pointers*, number 1, pages 193--204. ACM.

[Kaplan and Ullman, 1978] Kaplan, M. A. and Ullman, J. D. (1978). A general scheme for the automatic inference of variable types. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 60--75. ACM.

[Knapen et al., 1999] Knapen, G., Laguë, B., Dagenais, M., and Merlo, E. (1999). Parsing C++ despite missing declarations. In *IWPC*, pages 114--125. IEEE.

[Koppler, 1997] Koppler, R. (1997). A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637--649.

[Krebbers, 2015] Krebbers, R. (2015). *The C Standard Formalized in Coq*. PhD thesis, Radboud University Nijmegen.

[Krebbers and Wiedijk, 2015] Krebbers, R. and Wiedijk, F. (2015). A typed c11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 15--27. ACM.

[Larochelle et al., 2001] Larochelle, D., Evans, D., et al. (2001). Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, volume 32. Washington DC.

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258--282.

[McPeak and Necula, 2004] McPeak, S. and Necula, G. C. (2004). Elkhound: A fast, practical glr parser generator. In *International Conference on Compiler Construction*, pages 73--88. Springer.

[Memarian et al., 2016] Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R. N., and Sewell, P. (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*, pages 1--15. ACM.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348--375.

[Mitchell, 1991] Mitchell, J. C. (1991). Type inference with simple subtypes. *Journal of functional programming*, 1(3):245--285.

[Moonen, 2001] Moonen, L. (2001). Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13--22. IEEE.

[Mycroft, 1999] Mycroft, A. (1999). Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208--223. Springer.

[Naraschewski and Nipkow, 1999] Naraschewski, W. and Nipkow, T. (1999). Type inference verified: Algorithm w in isabelle/hol. *Journal of Automated Reasoning*, 23(3):299--318.

[Nazaré et al., 2014] Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., and Pereira, F. M. Q. (2014). Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791--809. ACM.

[Necula et al., 2002a] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. (2002a). Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213--228. Springer.

[Necula et al., 2002b] Necula, G. C., McPeak, S., and Weimer, W. (2002b). Ccured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128--139. ACM.

[Nielson et al., 2005] Nielson, F., Nielson, H. R., and Hankin, C. (2005). *Principles of program analysis*. Springer.

[Nienhuis et al., 2016] Nienhuis, K., Memarian, K., and Sewell, P. (2016). An operational semantics for c/c++11 concurrency. In *OOPSLA*, pages 111--128.

[Noonan et al., 2016] Noonan, M., Loginov, A., and Cok, D. (2016). Polymorphic type inference for machine code. In *PLDI*, pages 27--41. ACM.

[Odersky et al., 1999] Odersky, M., Sulzmann, M., and Wehr, M. (1999). Type inference with constrained types. *Theory and practice of object systems*, 5(1):35--55.

[Odersky et al., 2001] Odersky, M., Zenger, C., and Zenger, M. (2001). Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41--53.

[Padioleau, 2009] Padioleau, Y. (2009). Parsing C/C++ code without pre-processing. In *CC*, pages 109--125. Springer.

[Palsberg and Schwartzbach, 1991] Palsberg, J. and Schwartzbach, M. I. (1991). *Object-oriented type inference*, volume 26. ACM.

[Papaspyrou, 1998] Papaspyrou, N. S. (1998). *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens. Athens (Greece).

[Papaspyrou, 2001] Papaspyrou, N. S. (2001). Denotational semantics of ansi c. *Computer Standards & Interfaces*, 23(3):169--185.

[Perelman et al., 2012] Perelman, D., Gulwani, S., Ball, T., and Grossman, D. (2012). Type-directed completion of partial expressions. In *PLDI*, pages 275--286. ACM.

[Peyton Jones et al., 2003] Peyton Jones, S. et al. (2003). The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0--255.

[Peyton Jones et al., 2006] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. (2006). Simple unification-based type inference for gadts. In *ICFP*, volume 41, pages 50--61. ACM.

[Pierce, 2004] Pierce, B. C. (2004). *Types and Programming Languages.* MIT Press, 1st edition.

[Pierce and Turner, 2000] Pierce, B. C. and Turner, D. N. (2000). Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1--44.

[Pottier, 1996] Pottier, F. (1996). Simplifying subtyping constraints. In *ACM SIGPLAN Notices*, volume 31, pages 122--133. ACM.

[Pottier, 1998] Pottier, F. (1998). A framework for type inference with subtyping. In *ACM SIGPLAN Notices*, volume 34, pages 228--238. ACM.

[Pottier and Régis-Gianas, 2006] Pottier, F. and Régis-Gianas, Y. (2006). Stratified type inference for generalized algebraic data types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 232--244, New York, NY, USA. ACM.

[Pottier and Rémy, 2003] Pottier, F. and Rémy, D. (2003). The essence of ML type inference (preliminary version). Extended preliminary version of The Essence of ML Type Inference, in Advanced Topics in Types and Programming Languages.

[Pottier and Rémy, 2005] Pottier, F. and Rémy, D. (2005). The essence of ML type inference. In Pierce, B. C., editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389--489. MIT Press.

[Qt-Project, 2017] Qt-Project (2017). The qt creator ide. https://www.qt.io/ide/.

[Rémy, 1992] Rémy, D. (1992). Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France.

[Rémy, 2017] Rémy, D. (2017). Type systems for programming languages. http://pauillac.inria.fr/ remy/mpri/cours.pdf – accessed July, 2018.

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23--41.

[Rodrigues et al., 2013] Rodrigues, R. E., Campos, V. H. S., and Pereira, F. M. Q. (2013). A fast and low overhead technique to secure programs against integer overflows. In *CGO*, pages 1–11. ACM.

[Runtime-Verification, 2017] Runtime-Verification (2017). Rv-match. `https://runtimeverification.com/match/`.

[Sedgewick, 2002] Sedgewick, R. (2002). *Algorithms in C (3rd Edition).* Addison Wesley.

[Simonet, 2003] Simonet, V. (2003). Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Asian Symposium on Programming Languages and Systems*, pages 283--302. Springer.

[Smith and Volpano, 1996] Smith, G. and Volpano, D. (1996). Towards an ml-style polymorphic type system for c. In *European Symposium on Programming*, pages 341--355. Springer.

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

[Steensgaard, 1996] Steensgaard, B. (1996). Points-to analysis in almost linear time. In *POPL*, pages 32–41.

[Sterling, 1994] Sterling, L. (1994). *The Art of Prolog.* MIT Press, 2nd edition.

[Su et al., 2002] Su, Z., Aiken, A., Niehren, J., Priesnitz, T., and Treinen, R. (2002). *The first-order theory of subtyping constraints*, volume 37. ACM.

[Tillmann and De Halleux, 2008] Tillmann, N. and De Halleux, J. (2008). Pex: White box test generation for .net. In *TAP*, pages 134--153. Springer.

[Tiuryn and Wand, 1993] Tiuryn, J. and Wand, M. (1993). Type reconstruction with recursive types and atomic subtyping. In *Colloquium on Trees in Algebra and Programming*, pages 686--701. Springer.

[Ungar and Smith, 1987] Ungar, D. and Smith, R. B. (1987). *Self: The power of simplicity*, volume 22. ACM.

[Wadler and Findler, 2009] Wadler, P. and Findler, R. B. (2009). Well-typed programs can't be blamed. In *ESOP*, pages 1--16. Springer.

[Wand, 1987a] Wand, M. (1987a). Complete type inference for simple objects. *Symposium on Logic in Computer Science*, (2).

[Wand, 1987b] Wand, M. (1987b). A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115--121.

[Wand, 1988] Wand, M. (1988). Corrigendum: Complete type inference for simple objects. *Third Annual Symposium on Logic in Computer Science*.

[Williams et al., 2005] Williams, N., Marre, B., Mouy, P., and Roger, M. (2005). Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, pages 281--292. Springer.

Appendix A

Haskell Implementation of μC

This Appendix contains a Haskell implementation of the μC language that we introduced in Section 3.1. This source code, together with a a suite of test cases, can be obtained from PsycheC's GitHub repository, at `https://github.com/ltcmelo/psychec/tree/master/formalism`. The copyright of this source code, distributed under the terms of the GPLv3 license, is due to Leandro T. C. Melo. The reader is encouraged to read, compile, and run it. A great effort has been put into making this implementation very close to the formalism presented throughout this thesis. As a reminder, in order to employ our type inference technique on C programs, the PsycheC tool should be used instead. Besides being open-sourced in the aforementioned GitHub repository, an online interface to PsycheC is available at `http://cuda.dcc.ufmg.br/psyche-c/`.

A.1 The Syntax of μC

The implementation of μC' syntax, as presented in Figure 3.1.

```
data Stamp = Stamp Int deriving (Eq, Ord, Show)

newtype Ident = Ident { _x :: String } deriving (Eq, Ord, Show)

data Type = IntTy
          | DoubleTy
          | PtrTy Type
          | ConstTy Type
          | ArrowTy Type [Type]
          | RecTy [Decl] Ident
          | NamedTy Ident
          | TyVar Stamp
          deriving (Eq, Ord, Show)

data BinOptr = Add
             | Divide
             | Or
             | Assign
             deriving (Eq, Ord, Show)

data Lit = IntLit Int
         | DoubleLit Double
         deriving (Eq, Ord, Show)

data Expr = NumLit Lit
          | Var Ident
          | FldAcc Expr Ident
          | Deref Expr
          | AddrOf Expr
          | BinExpr BinOptr Expr Expr
          deriving (Eq, Ord, Show)

data Stmt = ExprStmt Expr
          | DeclStmt Decl
          | RetStmt Expr
          deriving (Eq, Ord, Show)

data Decl = Decl { _ft :: Type, _fx :: Ident } deriving (Eq,
   Ord, Show)

data FunDef = FunDef Type Ident [Decl] [Stmt] deriving (Eq,
   Ord, Show)

data TypeDef = TypeDef Type Type deriving (Eq, Ord, Show)

data Prog = Prog [TypeDef] [FunDef] deriving (Eq, Ord, Show)
```

A.2 The Constraints Language

The implementation of our constraints language, as presented in Figure 3.2.

```
data K = T
       | B
       | K :&: K
       | Exists [Type] K
       | Def Ident Type K
       | Fun Ident Type
       | TypeOf Ident Type
       | Syn Type Type
       | Has Type Decl
       | Type :=: Type
       | Type :<=: Type
       deriving (Eq, Ord, Show)
```

A.3 The Substitutions

The implementation of substitutions, as described in Section 3.3.

```
data Subst = Stamp :-> Type
           | Trivial
           deriving (Eq, Ord, Show)

class Substitutable a where
  -- | Apply a substitution.
  apply :: Subst -> a -> a
  -- | Convenience for multiple applications.
  applyMany :: [Subst] -> a -> a
  applyMany sl a = foldr (\s acc -> apply s acc) a sl
  -- | Get the free type variables.
  ftv :: a -> [Stamp]

instance Substitutable a => Substitutable [a] where
  apply s = map (apply s)
  ftv = foldr (union . ftv) []
```

```
instance Substitutable Type where
  apply Trivial t = t
  apply s t@(IntTy) = t
  apply s t@(DoubleTy) = t
  apply s (PtrTy t) = PtrTy (apply s t)
  apply s (ConstTy t) = ConstTy (apply s t)
  apply s (ArrowTy rt pt) = ArrowTy (apply s rt) (apply s pt)
  apply s (RecTy fs x) = RecTy (apply s fs) x
  apply s t@(NamedTy _) = t
  apply (st :-> t) t'@(TyVar st') = if st == st' then t else t'
  ftv IntTy = []
  ftv DoubleTy = []
  ftv (PtrTy t) = ftv t
  ftv (ConstTy t) = ftv t
  ftv (ArrowTy rt pt) = ftv rt `union` ftv pt
  ftv (RecTy fs _) = ftv fs
  ftv (NamedTy _) = []
  ftv (TyVar st) = [st]

instance Substitutable K where
  apply Trivial k = k
  apply _ T = T
  apply s (k1 :&: k2) = (apply s k1) :&: (apply s k2)
  apply s (Exists t k) = Exists (apply s t) (apply s k)
  apply s (Def x t@(TyVar _) k) = Def x (apply s t) (apply s k)
  apply s (Fun f t@(ArrowTy rt pl)) = Fun f (apply s t)
  apply s (TypeOf x t) = TypeOf x (apply s t)
  apply s (Syn t1 t2)  = Syn (apply s t1) (apply s t2)
  apply s (Has t fld) = Has (apply s t) (apply s fld)
  apply s (t1 :=: t2) = (apply s t1) :=: (apply s t2)
  apply s (t1 :<=: t2) = (apply s t1) :<=: (apply s t2)
  ftv _ = []

instance Substitutable Decl where
  apply s (Decl t x) = Decl (apply s t) x
  ftv (Decl t _) = ftv t

instance Substitutable v => Substitutable (Map k v) where
```

```
  apply s = Map.map (apply s)
  ftv = Map.foldr (union . ftv) []

-- This function exists for presentation purposes.
foreachValue :: Substitutable a => [Subst] -> Map k a -> Map k a
foreachValue s = Map.map (applyMany s)
```

A.4 The *type-id* of a Type

The implementation of function $\widehat{}$ (hat), as according to Definition 5.

```
newtype TypeId = TypeId { _id :: String } deriving (Eq, Ord,
   Show)

-- | Compute the typeid of a type.
hat :: Type -> TypeId
hat IntTy = TypeId "int"
hat DoubleTy = TypeId "double"
hat (PtrTy t) = TypeId $ (_id (hat t) ++ "*")
hat (ConstTy t) = TypeId $ "const " ++ (_id (hat t))
hat (ArrowTy rt pt) = TypeId $
  (_id (hat rt))  ++ "(*)(" ++
  (foldr (\t acc -> (_id (hat t)) ++ acc) ")" pt)
hat (RecTy _ x) = TypeId (_x x)
hat (NamedTy x) = TypeId (_x x)
hat (TyVar (Stamp n)) = TypeId $ "α" ++ (show n)
```

A.5 The Mappings ϕ, ψ, and Θ

The implementation of the map data structures and operations, according to Definition 6.

```
type Phi = Map Stamp Type
type Psi = Map Ident Type
type Theta = Map TypeId Type
```

```
-- | Find, in Psi, the type mapped to an identifier.
findInPsi :: Ident -> Psi -> Type
findInPsi x psi =
  case Map.lookup x psi of
    Just t -> t
    _ -> error $ "no τ for identifier " ++ show (ppK x) ++ " in
      ψ\n"

-- | Find, in Phi, the type mapped to a stamp.
findInPhi :: Stamp -> Phi -> Type
findInPhi st phi =
  case Map.lookup st phi of
    Just t -> t
    _ -> error $
         "no τ for stamp " ++ show (ppK st) ++ " in ϕ\n"

-- | Find, in Theta, the type mapped to a typeid.
findInTheta :: TypeId -> Theta -> Type
findInTheta hat theta =
  case Map.lookup hat theta of
    Just t -> t
    _ -> error $
         "no τ for typeid " ++ show (ppK hat) ++ " in Θ\n"

-- | Add, to Psi, an identifier to type mapping.
addToPsi :: Ident -> Type -> Psi -> Psi
addToPsi x t psi =
  case Map.lookup x psi of
    Just t -> error $
      show (ppK t) ++ ", indexed by " ++
      show (ppK x) ++ ", is already in ψ\n"
    _ -> Map.insert x t psi

-- | Add, to Phi, a stamp to type mapping.
addToPhi :: Stamp -> Type -> Phi -> Phi
addToPhi st t phi =
  case Map.lookup st phi of
    Just t -> error $
```

```
      show (ppK t) ++ ", indexed by " ++
      show (ppK st) ++ ", is already in ϕ\n"
    _ -> Map.insert st t phi

-- | Add, to Theta, a type id to type mapping.
addToTheta :: TypeId -> Type -> Theta -> Theta
addToTheta tid t theta =
  case Map.lookup tid theta of
    Just t -> error $
      show (ppK t) ++ ", indexed by " ++
      show (ppK tid) ++ ", is already in Θ\n"
    _ -> Map.insert tid t theta
```

A.6 The Semantics of Constraints

The implementation of our constraints semantics, as presented in Figure 3.4.

```
validateSemantics :: (Phi, Psi, Theta) -> K -> Bool
validateSemantics (phi, psi, theta) k = satisfies (phi, psi,
   theta) k

satisfies :: (Phi, Psi, Theta) -> K -> Bool

-- | KT
satisfies (_, _, _) T = True

-- | KAnd
satisfies (phi, psi, theta) (k1 :&: k2) =
  let check1 = satisfies (phi, psi, theta) k1
      check2 = satisfies (phi, psi, theta) k2
  in if trace_Sema
     then trace ("[trace Sema] " ++ show (ppK k1)) check1
          && trace ("[trace Sema] " ++ show (ppK k2)) check2
     else check1 && check2

-- | KEx
satisfies (phi, psi, theta) kk@(Exists tl k) =
  let f st = isGround (findInPhi st phi)
  in foldr (\(TyVar st) acc -> f st && acc) True tl
     && satisfies (phi, psi, theta) k

-- | KDef
satisfies (phi, psi, theta) (Def x (TyVar st) k) =
  findInPsi x psi == findInPhi st phi
  && satisfies (phi, psi, theta) k

-- | KFun
satisfies (phi, psi, theta) (Fun f (ArrowTy rt@(TyVar st) p) k)
   =
  findInPsi f psi == ArrowTy (findInPhi st phi) p
  && satisfies (phi, psi, theta) k

-- | KInst
satisfies (phi, psi, theta) k@(TypeOf x t@(TyVar _)) =
  satisfies (phi, psi, theta) ((findInPsi x psi) :=: t)

-- | KSyn
satisfies (phi, psi, theta) (Syn t a@(TyVar _)) =
  let t' = findInTheta (hat t) theta
  in satisfies (phi, psi, theta) (t' :=: a)

-- | KHas
satisfies (phi, psi, theta) (Has (TyVar st) (Decl t x)) =
  let tt = findInPhi st phi
      t' = field x (findInTheta (hat tt) theta)
  in satisfies (phi, psi, theta) (t' :=: t)

-- | KEq
satisfies (phi, _, _) k@(t1 :=: t2) = isSubTy phi t1 t2

-- | KIq
satisfies (phi, _, _) k@(t1 :<=: t2) = isSubTy phi t1 t2

trace_Sema = False

-- | Return the type of the field in a record.
```

```
field :: Ident -> Type -> Type
field x (RecTy ds _) =
  let ds' = filter (\(Decl _ x') -> x == x') ds
  in case length ds' of
    1 -> _ft (ds' !! 0)
    _ -> error $ "record has no field " ++ show (ppK x) ++ "\n"
field _ t = error $ "type " ++ show (ppK t) ++ " is not a
   record\n"

-- | Return whether the type is a ground type.
isGround :: Type -> Bool
isGround t =
  if ftv t == []
  then True
  else False

-- | Supporting function to check entailment across rewrites.
checkEntail :: (Config, Config) -> K -> Config
checkEntail (cfg, cfg') k =
  if validateSemantics ((phi cfg), (psi cfg), (theta cfg)) k
  then cfg'
  else error $ "entailment failed"
```

A.7 The Type Predicate of Constraints Semantics

The implementation of our type predicate, as presented in Figure 3.5.

```
isSubTy :: Phi -> Type -> Type -> Bool
isSubTy phi t1@(TyVar st1) t2 =
  if (isIdentity phi t1)
  then True
  else (isGround (findInPhi st1 phi))
          && isSubTy phi (findInPhi st1 phi) t2
isSubTy phi t1 t2@(TyVar st2) =
  if (isIdentity phi t2)
  then True
  else (isGround (findInPhi st2 phi))
```

```
    && isSubTy phi t1 (findInPhi st2 phi)
isSubTy phi (ConstTy t1) (ConstTy t2) =
  isSubTy phi t1 t2
isSubTy phi (ConstTy t1) t2 =
  isSubTy phi t1 t2
isSubTy phi t@(PtrTy t1) t'@(PtrTy t2) =
  isSubTyPtr phi t1 t2
isSubTy _ IntTy IntTy = True
isSubTy _ DoubleTy DoubleTy = True
isSubTy _ IntTy DoubleTy = True
isSubTy _ (NamedTy x1) (NamedTy x2) = x1 == x2
isSubTy _ t1@(RecTy _ _) t2@(RecTy _ _) = t1 == t2
isSubTy phi t1 t2 =
  error $ "unknown (value) subtyping relation " ++
  show (ppK t1) ++ "<:" ++ show (ppK t2)

isSubTyPtr :: Phi -> Type -> Type -> Bool
isSubTyPtr phi t1@(TyVar st1) t2 =
  if (isIdentity phi t1)
  then True
  else (isGround (findInPhi st1 phi))
    && isSubTyPtr phi (findInPhi st1 phi) t2
isSubTyPtr phi t1 t2@(TyVar st2) =
  if (isIdentity phi t2)
  then True
  else (isGround (findInPhi st2 phi))
    && isSubTyPtr phi t1 (findInPhi st2 phi)--}
isSubTyPtr phi (ConstTy t1) (ConstTy t2) =
  isSubTyPtr phi t1 t2
isSubTyPtr phi t1 (ConstTy t2) =
  isSubTyPtr phi t1 t2
isSubTyPtr phi (PtrTy t1) (PtrTy t2) =
  isSubTyPtr phi t2 t2
isSubTyPtr _ IntTy IntTy = True
isSubTyPtr _ DoubleTy DoubleTy = True
isSubTyPtr _ (NamedTy x1) (NamedTy x2) = x1 == x2
isSubTyPtr _ t1@(RecTy _ _) t2@(RecTy _ _) = t1 == t2
isSubTyPtr phi t1 t2 =
  error $ "unknown (reference) subtyping relation " ++
  show (ppK t1) ++ "<:" ++ show (ppK t2)


-- | Subtyping predicate for ground types.

isSubTy' :: Type -> Type -> Bool
isSubTy' (TyVar _) _ = error $ "expected ground type "
isSubTy' _ (TyVar _) = error $ "expected ground type "
isSubTy' (ConstTy t1) (ConstTy t2) =
  isSubTy' t1 t2
isSubTy' (ConstTy t1) t2 =
  isSubTy' t1 t2
isSubTy' (PtrTy t1) (PtrTy t2) =
  isSubTyPtr' t1 t2
isSubTy' IntTy IntTy = True
isSubTy' DoubleTy DoubleTy = True
isSubTy' IntTy DoubleTy = True
isSubTy' (NamedTy x1) (NamedTy x2) = x1 == x2
isSubTy' t1@(RecTy _ _) t2@(RecTy _ _) = t1 == t2
isSubTy' t1 t2 =
  error $ "unknown (value/ground) subtyping relation " ++
  show (ppK t1) ++ "<:" ++ show (ppK t2)

isSubTyPtr' :: Type -> Type -> Bool
isSubTyPtr' (TyVar _) _ = error $ "expected ground type "
isSubTyPtr' _ (TyVar _) = error $ "expected ground type "
isSubTyPtr' (ConstTy t1) (ConstTy t2) =
  isSubTyPtr' t1 t2
isSubTyPtr' t1 (ConstTy t2) =
  isSubTyPtr' t1 t2
isSubTyPtr' (PtrTy t1) (PtrTy t2) =
  isSubTyPtr' t2 t2
isSubTyPtr' IntTy IntTy = True
isSubTyPtr' DoubleTy DoubleTy = True
isSubTyPtr' (NamedTy x1) (NamedTy x2) = x1 == x2
isSubTyPtr' t1@(RecTy _ _) t2@(RecTy _ _) = t1 == t2
isSubTyPtr' t1 t2 =
  error $ "unknown (reference/ground) subtyping relation " ++
  show (ppK t1) ++ "<:" ++ show (ppK t2)
```

```
-- | Whether we have an identity relation.
isIdentity phi t@(TyVar st) = t == findInPhi st phi
```

A.8 The Constraint Generators

The implementation of the constraint generators from Figure 3.6.

```
generateConstraints :: Prog -> M -> IO K
generateConstraints p m = do
  (c, _) <- runStateT (genProg p m) 0
  return c

-- | The generator is typed as a monad to allow isolation of
-- the fresh variable supply.
type GenMonad a = StateT Int IO a
fresh :: GenMonad Type
fresh = do
  n <- get
  put (n + 1)
  return $ TyVar (Stamp n)

-- | Constraint generation for a program.
genProg :: Prog -> M -> GenMonad K
genProg (Prog _ []) _ = return T
genProg (Prog [] ((FunDef rt f d s):fs)) m = do
  a <- fresh
  syn <- buildSyn rt a
  let pt = foldl (\acc (Decl t _) -> t:acc) [] d
  k <- genFun d s a m
  k' <- genProg (Prog [] fs) m
  return $
    Exists [a] $
    syn :&:
    Fun f (ArrowTy a pt) k :&:
    k'
genProg (Prog ((TypeDef t1 t2):tds) fs) m = do
```

```
  a <- fresh
  k <- genProg (Prog tds fs) m
  return $
    Exists [a] $
    (Syn t2 a) :&:
    (a :=: t1) :&:
    k

-- | Constraint generation for functions.
genFun :: [Decl] -> [Stmt] -> Type -> M -> GenMonad K
genFun [] s rt m = genStmt s rt m
genFun ((Decl t x):dx) s rt m = do
  a <- fresh
  syn <- buildSyn t a
  k <- genFun dx s rt m
  return $
    Exists [a] $
    syn :&:
    Def x a k

-- | Constraint generation for statements.
genStmt :: [Stmt] -> Type -> M -> GenMonad K
genStmt ((DeclStmt (Decl t x)):sl) rt m = do
  a <- fresh
  syn <- buildSyn t a
  k <- genStmt sl rt m
  return $
    Exists [a] $
    syn :&:
    Def x a k
genStmt ((ExprStmt e):sl) rt m = do
  a <-  fresh
  k1 <- genExpr e a m
  k2 <- genStmt sl rt m
  return $
    Exists [a] k1 :&:
    k2
genStmt ((RetStmt e):[]) rt m = do
  a <- fresh
  k <- genExpr e a m
  return $
    Exists [a] $
    keepOrDrop (shape (Var (Ident "$ret")) m) rt (shape e m) a
       Assign :&:
    k

-- | Constraint generation for expressions.
genExpr :: Expr -> Type -> M -> GenMonad K
genExpr (NumLit l) t _ = return (rho l :=: t)
genExpr (Var x) t _ = return (TypeOf x t)
genExpr (FldAcc e x) t m = do
  a1 <- fresh
  a2 <- fresh
  a3 <- fresh
  k <- genExpr e a1 m
  return $
    Exists [a1, a2, a3] $
    (Has a2 (Decl a3 x)) :&:
    (a1 :=: (PtrTy a2)) :&:
    (a3 :=: t) :&:
    k
genExpr (Deref e) t m = do
  a <- fresh
  k <- genExpr e a m
  return $
    Exists [a] $
    (a :=: PtrTy t) :&:
    k
genExpr (AddrOf e) t m = do
  a1 <- fresh
  a2 <- fresh
  k <- genExpr e a2 m
  return $
    Exists [a1, a2] $
    (a1 :=: PtrTy a2) :&:
    (a1 :=: t) :&:
    k
genExpr e@(BinExpr op e1 e2) t m = do
```

```
  a1 <- fresh
  a2 <- fresh
  k1 <- genExpr e1 a1 m
  k2 <- genExpr e2 a2 m
  return $
    Exists [a1, a2] $
    k1 :&:
    k2 :&:
    keepOrDrop (shape e1 m) a1 (shape e2 m) a2 op :&:
    select (shape e1 m) a1 (shape e2 m) a2 t op

-- | The type of a literal.
rho :: Lit -> Type
rho (IntLit _) = IntTy
rho (DoubleLit _) = DoubleTy
```

A.9 The *build synonym* Auxiliary Generator

The implementation of auxiliary generator *build synonym*, as presented in Figure 3.7.

```
-- | Recursively build type synonyms.
buildSyn :: Type -> Type -> GenMonad K
buildSyn t@(PtrTy tt) a = do
  b <- fresh
  syn <- buildSyn tt b
  return $
    Exists [b] $
    Syn t a :&:
    Syn tt b :&:
    ((PtrTy b) :=: a) :&:
    syn
buildSyn t@(ConstTy tt) a = do
  b <- fresh
  syn <- buildSyn tt b
  return $
    Exists [b] $
    Syn t a :&:
```

```
    Syn tt b :&:
    ((ConstTy b) :=: a) :&:
    syn
buildSyn t a =
  return $ Syn t a
```

A.10 The Classification of Expressions

The implementation for the classification of expressions to shapes of our lattice $\mathcal{L}$, corresponding to the rules in Figure 3.9.

```
data Shape = Undefined
           | Scalar
           | Pointer
           | Numeric
           | Integral
           | Floating
           deriving (Eq, Ord, Show)

newtype M = M { _shapes :: Map Expr Shape } deriving (Eq, Ord,
   Show)

classifyE :: Expr -> Shape -> M -> (Shape, M)
classifyE e@(NumLit v) _ m =
  insertOrUpdate e sp m
  where
    sp = case v of
      (IntLit 0) -> Scalar
      (IntLit _) -> Integral
      _ -> Floating
classifyE e@(Var _) sp m =
  insertOrUpdate e sp m
classifyE e@(FldAcc e' x) sp m =
  insertOrUpdate e sp m'
  where
    (_, m') = classifyE e' Pointer m
classifyE e@(Deref e') sp m =
  insertOrUpdate e sp m'
  where
    (_, m') = classifyE e' Pointer m
classifyE e@(AddrOf e') sp m =
  insertOrUpdate e Pointer m'
  where
    (_, m') = classifyE e' sp m
classifyE e@(BinExpr Add e1 e2) sp m =
  insertOrUpdate e sp'''' m''
  where
    sp' = if (sp == Integral
              || sp == Floating
              || sp == Numeric)
          then sp
          else Scalar
    (sp1, m') = classifyE e1 sp' m
    sp'' = if (sp1 == Pointer)
           then Integral
           else if (sp == Integral
                    || sp == Floating
                    || sp == Numeric)
                then sp
                else Scalar
    (sp2, m'') = classifyE e2 sp'' m'
    sp''' = if (sp2 == Pointer)
            then Integral
            else sp''
    (sp3, m''') = classifyE e1 sp''' m''
    sp'''' = if (sp3 == Pointer || sp2 == Pointer)
             then Pointer
             else Numeric
classifyE e@(BinExpr Divide e1 e2) sp m =
  insertOrUpdate e Numeric m''
  where
    (_, m') = classifyE e1 Numeric m
    (_, m'') = classifyE e2 Numeric m'
classifyE e@(BinExpr Or e1 e2) sp m =
  insertOrUpdate e Integral m''
  where
```

```
    (_, m') = classifyE e1 Scalar m
    (_, m'') = classifyE e2 Scalar m'
classifyE e@(BinExpr Assign e1 e2) sp m =
  insertOrUpdate e sp1 m''
  where
    (sp2, m') = classifyE e2 sp m
    (sp1, m'') = classifyE e1 sp2 m'

classifyD :: Decl -> M -> (Shape, M)
classifyD (Decl { _ft = t, _fx = x }) m =
   insertOrUpdate (Var x) (ty2shape t) m



insertOrUpdate :: Expr -> Shape -> M -> (Shape, M)
insertOrUpdate e sp m =
  (sp', M $ Map.insert e sp' (_shapes m))
  where
    sp' = case Map.lookup e (_shapes m) of
      Just sp'' ->
        case sp'' of
          Pointer -> sp''
          Integral -> sp''
          Floating -> sp''
          Numeric ->
            if (sp == Integral || sp == Floating)
            then sp
            else sp''
          Scalar ->
            if (sp == Integral
                || sp == Floating
                || sp == Pointer)
            then sp
            else sp''
          Undefined -> sp
      Nothing -> sp

shape :: Expr -> M -> Shape
shape e m =
  case Map.lookup e (_shapes m) of
```

```
    Just sp -> sp
    Nothing -> Undefined

ty2shape :: Type -> Shape
ty2shape IntTy = Integral
ty2shape DoubleTy = Floating
ty2shape (ConstTy t) = ty2shape t
ty2shape (PtrTy _) = Pointer
ty2shape _ = Undefined

-- | Build lattice of shapes until stabilization.
buildLattice :: Prog -> M -> M
buildLattice p@(Prog _ fs) m =
  let
    go ((DeclStmt d):xs) acc =
      let (sp, m) = classifyD d acc
      in go xs m
    go ((ExprStmt e):xs) acc =
      let (sp, m) = classifyE e (shape e acc) acc
      in go xs m
    go ((RetStmt e):[]) acc = snd $ classifyE e (shape e acc)
       acc

    handleParam ds = map (\d -> DeclStmt d) ds
    handleRet rt m = M $ Map.insert (Var (Ident "$ret"))
       (ty2shape rt) (_shapes m)

    m' = foldr (\(FunDef rt _ ds ss) acc -> go ((handleParam
       ds) ++ ss)
               (handleRet rt acc)) m fs
  in if (m' == m)
     then m'
     else buildLattice p m'
```

A.11 The *keep or drop* and *select* Auxiliary Generators

The implementation of auxiliary generators *keep or drop* and *select*, as presented in Figure 3.10.

```
-- | Keep or drop a constraint.
keepOrDrop :: Shape -> Type -> Shape -> Type -> BinOptr -> K
keepOrDrop sp1 a1 sp2 a2 op =
  if (sp1 /= sp2
      && (sp1 == Pointer || sp2 == Pointer)
      && sp1 /= Undefined
      && sp2 /= Undefined)
  then T
  else if (op == Assign)
       then (a2 :<=: a1)
       else if (sp1 == Integral && sp2 == Floating)
            then (a1 :<=: a2)
            else if (sp1 == Floating && sp2 == Integral)
                 then (a2 :<=: a1)
                 else (a1 :=: a2)

-- | Select operands and result types.
select :: Shape -> Type -> Shape -> Type -> Type -> BinOptr -> K
select sp1 a1 sp2 a2 t op  =
  case op of
    Add -> if (sp1 == Pointer)
           then (a1 :=: t) :&: (a2 :=: IntTy)
           else if (sp2 == Pointer)
                then (a2 :=: t) :&: (a1 :=: IntTy)
                else (t :<=: DoubleTy)
    Assign -> (t :=: a1)
    Or -> (t :=: IntTy)
    Divide -> if (sp1 == Integral && sp2 == Integral)
              then (t :=: IntTy)
                   :&: (a1 :=: IntTy)
                   :&: (a2 :=: IntTy)
              else if (sp1 == Integral)
```

```
                    then (t :<=: DoubleTy)
                         :&: (a1 :=: IntTy)
                         :&: (a2 :<=: DoubleTy)
                    else if (sp2 == Integral)
                         then (t :<=: DoubleTy)
                              :&: (a1 :<=: DoubleTy)
                              :&: (a2 :=: IntTy)
                         else (t :<=: DoubleTy)
                              :&: (a1 :<=: DoubleTy)
                              :&: (a2 :<=: DoubleTy)
```

A.12 The Unifications Algorithms $\mathcal{U}_c$ and $\mathcal{U}_s$

The implementation of our unification algorithms, as presented in Figure 4.2.

```
data StratMode = Relax
               | Enforce
               deriving (Eq, Ord, Show)

class Substitutable a => UnifiableC a where
  uC :: a -> a -> [Subst]
  uS :: a -> a -> StratMode -> [Subst]

instance UnifiableC Type where
  uC (TyVar st) t2 =
    let s = st :-> t2
    in if (trace_UC)
       then trace("[trace uC] " ++ show (ppK s)) [s]
       else [s]
  uC t1 t2@(TyVar _) = uC t2 t1
  uC IntTy IntTy = [Trivial]
  uC DoubleTy DoubleTy = [Trivial]
  uC t1@(NamedTy x1) t2@(NamedTy x2)
    | x1 == x2 = [Trivial]
    | otherwise = error $ "can't unify named types " ++
                  (show $ ppK t1) ++ "::" ++ (show $ ppK t2)
  uC (ConstTy t1) (ConstTy t2) = uC t1 t2
  uC (PtrTy t1) (PtrTy t2) = uC t1 t2
  uC t1@(RecTy fs1 x1) t2@(RecTy fs2 x2) = undefined
  uC (ArrowTy rt1 [pt1]) (ArrowTy rt2 [pt2]) = undefined
  uC t1 t2 = error $ "unknown unification from " ++
             (show $ ppK t1) ++ " to " ++ (show $ ppK t2)


  uS t1@(PtrTy _) (TyVar st) _ = [st :-> t1]
  uS t1 t2@(TyVar st) sm
    | sm == Enforce = [st :-> t1]
    | otherwise = [st :-> (relax t1)]
  uS (TyVar st) t2 _ = [st :-> (relax t2)]
  uS IntTy IntTy _ = [Trivial]
  uS IntTy DoubleTy Relax = [Trivial]
  uS DoubleTy DoubleTy _ = [Trivial]
  uS t1@(NamedTy x1) t2@(NamedTy x2) _
    | x1 == x2 = [Trivial]
    | otherwise = error $ "can't unify named types " ++
                  (show $ ppK t1) ++ "::" ++ (show $ ppK t2)
  uS (ConstTy t1) (ConstTy t2) sm = uS t1 t2 sm
  uS (ConstTy t1) t2 Relax = uS t1 t2 Relax
  uS t1 (ConstTy t2) _ = uS t1 t2 Relax
  uS (PtrTy t1) (PtrTy t2) _ = uS t1 t2 Enforce
  uS t1@(RecTy fs1 x1) t2@(RecTy fs2 x2) _ = undefined
  uS (ArrowTy rt1 [pt1]) (ArrowTy rt2 [pt2]) _ = undefined
  uS t1 t2 _ = error $ "unknown unification from " ++
               (show $ ppK t1) ++ " to " ++ (show $ ppK t2)

instance UnifiableC Decl where
  uC (Decl t1 x1) (Decl t2 x2)
    | x1 == x2 = uC t1 t2
    | otherwise = error $ "can't unify decl " ++
          (show x1) ++ "::" ++ (show x2)
  uS (Decl t1 x1) (Decl t2 x2) m
    | x1 == x2 = uS t1 t2 m
    | otherwise = error $ "can't unify decl " ++
          (show x1) ++ "::" ++ (show x2)

instance UnifiableC a => UnifiableC [a] where
```

```
  uC [] [] = [Trivial]
  uC _  [] = error "can't unify lists, different lengths"
  uC [] x  = uC x []
  uC (a1:as1) (a2:as2) =
    let s = uC a1 a2
        s' = uC (applyMany s as1) (applyMany s as2)
    in s ++ s'

  uS [] [] _ = [Trivial]
  uS _  [] _ = error "can't unify lists, different lengths"
  uS [] x m = uS x [] m
  uS (a1:as1) (a2:as2) m =
    let s = uS a1 a2 m
        s' = uS (applyMany s as1) (applyMany s as2) m
    in s ++ s'

trace_UC = False
trace_US = False
```

A.13 The *relax* Function

The implementation of the `const` *relax* function, used by unification $\mathcal{U}_s$ and presented in Figure 4.3.

```
-- | Relax constness.
relax :: Type -> Type
relax (ConstTy t) = t
relax (PtrTy t) = PtrTy (relax t)
relax t = t
```

A.14 The Solver Configuration and Driver

The implementation of our solver's configuration, as described in Section 4.2, along with the solver driver.

```
data Config = Config
  { phi :: Phi,
    psi :: Psi,
    theta :: Theta,
    k :: K,
    kE :: [K],
    kI :: [K],
    kW :: [K],
    kF :: [K]
  }
  deriving (Eq, Ord, Show)
```

A.15 The Preprocessing Stage of the Solving Process

The implementation of the preprocessing rules from Figure 4.5.

```
preprocess :: Config -> Config

-- | PP-and
preprocess cfg@(Config { k = k1 :&: k2 }) =
  let cfg' = preprocess (cfg { k = k1 })
  in  preprocess $ (cfg' { k = k2 })

-- | PP-ex
preprocess cfg@(Config { k = Exists ts k' }) =
  let self t@(TyVar st) acc = addToPhi st t acc
      phi' = foldr self (phi cfg) ts
  in preprocess $ cfg { phi = phi', k = k' }

-- | PP-syn
preprocess cfg@(Config { k = Syn t a, theta } ) =
  let theta' = if Map.member (hat t) theta
               then theta
               else addToTheta (hat t) a theta
  in preprocess $ cfg { theta = theta',
                        k = ((findInTheta (hat t) theta') :=:
                           a) }

-- | PP-def
preprocess cfg@(Config { k = Def x (TyVar st) k' }) =
  let psi' = addToPsi x (findInPhi st (phi cfg)) (psi cfg)
  in preprocess $ cfg { psi = psi', k = k' }

-- | PP-fun
preprocess cfg@(Config { k = Fun f (ArrowTy (TyVar st) t) k' })
   =
  let psi' = addToPsi f (ArrowTy (findInPhi st (phi cfg)) t)
     (psi cfg)
  in preprocess $ cfg { psi = psi', k = k' }

-- | PP-inst
preprocess cfg@(Config { k = TypeOf x t }) =
  let k' = findInPsi x (psi cfg) :=: t
  in preprocess $ cfg { k = k' }

-- | PP-eq
preprocess cfg@(Config { k = k'@(t1 :=: t2), kE }) =
  preprocess $ cfg { k = T, kE = k':kE }

-- | PP-has
preprocess cfg@(Config { k = k'@(Has _ _), kF }) =
  preprocess $ cfg { k = T, kF = k':kF }

-- | PP-iq
preprocess cfg@(Config { k = k'@(t1 :<=: t2), kI } ) =
  preprocess $ cfg { k = T, kI = k':kI }

-- | PP-end
preprocess cfg@(Config { k = T }) =
  if (not trace_PP)
  then cfg
  else trace ("[trace PP]\n" ++ showConfig cfg ++ "\n") cfg

trace_PP = False
```

A.16 The 1^{st} Unification Round of the Solving Process

The implementation of the 1^{st} unification round rule from Figure 4.6.

```
trace_U = False

unifyEq :: Config -> Config

-- | UE-base
unifyEq cfg@(Config { kE = k@(t1 :=: t2):kE_ }) =
  let s = uC t1 t2
      phi' = foreachValue s (phi cfg)
      psi' = foreachValue s (psi cfg)
      theta' = foreachValue s (theta cfg)
      kE' = applyMany s kE_
      kF' = applyMany s (kF cfg)
      kI' = applyMany s (kI cfg)
      cfg' = cfg { phi = phi',
                   psi = psi',
                   theta = theta',
                   kE = kE',
                   kF = kF',
                   kI = kI' }
      rw = unifyEq (checkEntail (cfg, cfg') k)
  in if (not trace_U)
     then rw
     else trace("[uni-eq]: " ++ show (ppK s)) rw
```

A.17 Functions *splitWob* and *orderSub*

The implementation of functions *splitWob* and *orderSub* from Figure 4.7.

```haskell
-- | Order inequality constraints.
orderSub :: [K] -> [K] -> [K]
orderSub ((t1@(PtrTy (ConstTy _)) :<=: t2):kW) kS =
  orderSub kW ((t1 :<=: t2):kS)
orderSub ((t1 :<=: t2@(PtrTy _)):kW) kS =
  orderSub kW ((t1 :<=: t2):kS)
orderSub ((t1@DoubleTy :<=: t2):kW) kS =
  orderSub kW ((t1 :<=: t2):kS)
orderSub ((t1 :<=: t2@IntTy):kW) kS =
  orderSub kW ((t1 :<=: t2):kS)
orderSub ((t1 :<=: t2):kW) kS =
  orderSub (kW ++ [t1 :<=: t2]) kS
orderSub (B:kW) kS =
  kS ++ kW

-- | Split wobbly relations.
splitWob :: [K] -> [K] -> ([K], [K])
splitWob (w@((TyVar _) :<=: (TyVar _)):k) kW =
  splitWob k (w:kW)
splitWob (nw@(t1 :<=: t2):k) kW =
  splitWob (k ++ [nw]) kW
splitWob (B:k) kW =
  (k, kW)
```

A.18 The 2^{nd} Unification Round of the Solving Process

The implementation of the 2^{nd} unification round rules from Figure 4.8.

```haskell
unifyIq :: Config -> Config

-- | SO
splitOrder cfg =
  let (kI', kW') = splitWob ((kI cfg) ++ [B]) []
      kI'' = orderSub (kI' ++ [B]) []
  in cfg { kI = kI'', kW = kW'}
```

```haskell
-- | UI-base
unifyIq cfg@(Config {
                kI = k@(t1 :<=: t2):kI_ }) =
  let s = uS t1 t2 Relax
      phi' = foreachValue s (phi cfg)
      psi' = foreachValue s (psi cfg)
      theta' = foreachValue s (theta cfg)
      kI' = applyMany s kI_
      kF' = applyMany s (kF cfg)
      kW' = applyMany s (kW cfg)
      (kI'', kW'') = splitWob (kI' ++ kW' ++ [B]) []
      kI''' = orderSub (kI'' ++ [B]) []
      cfg' = cfg { phi = phi',
                   psi = psi',
                   theta = theta',
                   kI = kI''',
                   kF = kF',
                   kW = kW'' }
      rw = unifyIq (checkEntail (cfg, cfg') k)
  in if (not trace_U)
     then rw
     else trace("[uni-iq]: " ++ show (ppK s)) rw

-- | UI-end
unifyIq cfg = cfg


-- | UW-base
unifyWb ::  Config -> Config
unifyWb cfg@(Config { kI = k@(t1 :<=: t2):kI_ }) =
  let s = uS t1 t2 Relax
      phi' = foreachValue s (phi cfg)
      psi' = foreachValue s (psi cfg)
      theta' = foreachValue s (theta cfg)
      kI_' = applyMany s kI_
      kF' = applyMany s (kF cfg)
      cfg' = cfg { phi = phi',
                   psi = psi',
                   theta = theta',
                   kI = kI_',
                   kF = kF' }
      rw = unifyWb (checkEntail (cfg, cfg') k)
  in if (not trace_U)
     then rw
     else trace("[uni-w]: " ++ show (ppK s)) rw

-- | UW-end
unifyWb cfg = cfg
```

A.19 The Membership Normalization Stage of the Solving Process

The implementation of the rules from Figure 4.9.

```
-- | SH
sortHas :: Config -> Config
sortHas cfg =
  let criteria k1@(Has t1 (Decl _ x1)) k2@(Has t2 (Decl _ x2))
        | t1 == t2 = compare x1 x2
        | otherwise = compare t1 t2
  in cfg { kF = sortBy criteria (kF cfg) }

-- | MN-join
normFlds :: Config -> Config
normFlds cfg@(Config {
                kE,
                kF = h@(Has t1 (Decl ft1 x1)):kF_@(Has t2 (Decl
                   ft2 x2):_) }) =
  let kE' = if (t1 == t2) && (x1 == x2)
             then (ft1 :=: ft2):kE
             else kE
  in normFlds cfg { kE = kE', kF = kF_ ++ [h] }

-- | MN-skip
normFlds cfg@(Config { kF = h@(Has _ _):B:kF_ }) =
  cfg { kF = kF_ ++ [h] }

-- | MN-nfld
normFlds cfg@(Config { kE = [], kF = [B] }) =
  cfg { kF = [] }
```

A.20 The Field Convergence Algorithm

The implementation of the convergence algorithm from Figure 4.10.

```
-- | Convergence of has constraints
converge :: Config -> IO Config
converge cfg = do
  let cfg' = sortHas cfg
  debug "sort-membership" (showConfig cfg')

  let cfg'' = normFlds cfg' { kF = (kF cfg') ++ [B] }
  debug "equalize-fields" (showConfig cfg'')

  let cfg''' = unifyEq cfg''
  debug "unify-fields" (showConfig cfg''')

  if (kF cfg'') == (kF cfg''')
    then return $ cfg'''
    else converge cfg'''
```

A.21 The Record Composition Stage of the Solving Process

The implementation of the record composition rules from Figure 4.11.

```
composeRecs :: Config -> Config

-- | RC-inst
```

```
composeRecs cfg@(Config {
                    phi,
                    psi,
                    theta,
                    kF = k@(Has t@(TyVar st@(Stamp n)) d):kF_
                       }) =
  let x = Ident $ "TYPE_" ++ (show n)
      t = NamedTy x
      s = st :-> t
      theta' = (addToTheta (hat t) (RecTy [d] x) theta)
  in composeRecs cfg { phi = apply s phi,
                       psi = apply s psi,
                       theta = apply s theta',
                       kF = apply s kF_ }

-- | RC-upd
composeRecs cfg@(Config {
                    theta,
                    kF = k@(Has t@(NamedTy _) d):kF_ }) =
  case findInTheta (hat t) theta of
    r@(RecTy dl x) ->
      let r' = if (elem d dl) then r else (RecTy (d:dl) x)
      in composeRecs cfg { kF = kF_,
                           theta = getsUpdate (hat t) r' theta }
    _ -> error $ "can't recognized record " ++ show (ppK k)

-- | RC-end
composeRecs cfg = cfg



-- | Update the type id to type mapping of Theta.
getsUpdate :: TypeId -> Type -> Theta -> Theta
getsUpdate tid t theta =
  case Map.lookup tid theta of
    Just _ -> Map.alter (\_ -> Just t) tid theta
    _ -> error $ show (ppK tid) ++ " can't be found in ϕ for
       update\n"
```

A.22 The De-orphanization Stage of the Solving Process

The implementation of the de-orphanization rule from Figure 4.12.

```
deorph :: Config -> Config

-- | DO
deorph cfg@Config { phi, psi, theta } =
  cfg { phi = phi', psi = psi', theta = theta' }
  where
    bind ((tid, TyVar st@(Stamp n)):l) =
      if "struct " `isPrefixOf` (_id tid)
      then (st :-> (RecTy [Decl IntTy (Ident "dummy")] (Ident
         (_id tid)))):(bind l)
      else (st :-> NamedTy (Ident "int/*orphan*/ ")):(bind l)
    bind (_:l) = bind l
    bind [] = []

    s = bind (Map.toList theta)
    phi' = foreachValue s phi
    psi' = foreachValue s psi
    theta' = foreachValue s theta
```

A.23 The Complete Solver Algorithm

The implementation of the complete solver procedure from Figure 4.13.

```
solveConstraints :: K -> Config -> IO Config
solveConstraints k cfg = do
  let cfgPP = preprocess cfg
  debug "preprocessing" (showConfig cfgPP)

  let cfgUE = unifyEq cfgPP
  debug "unify-equivalences" (showConfig cfgUE)
  let cfgSO = splitOrder cfgUE
  debug "split-order-inequalities" (showConfig cfgSO)

  let cfgUI = unifyIq cfgSO
  debug "unify-inequalities" (showConfig cfgUI)

  let cfgUP = unifyWb cfgUI { kI = (kW cfgUI), kW = [] }
  debug "unify-wobbly" (showConfig cfgUP)

  cfgUF <- converge cfgUP

  let cfgCR = composeRecs cfgUF
  debug "compose-records" (showConfig cfgCR)

  let cfgOR = deorph cfgCR
  debug "deorph" (showConfig cfgOR)

  return cfgOR

debug msg content = do
  putStrLn $ "\n<<< " ++ msg ++ " >>>\n" ++ content
  writeFile (msg ++ ".log") content
```

A.24 The Typing Rules

The implementation of the typing rules, as presented in Figure 4.15.

```
type Gamma = Map Ident Type

verifyTyping :: Config -> Gamma -> Prog -> [Type]
verifyTyping c g p = typeProg c g p

-- | TCPrg
typeProg :: Config -> Gamma -> Prog -> [Type]
typeProg c gam (Prog _ fl) =
  foldr (\f acc -> (typeFunDef c gam f):acc) [] fl
```

```
-- | TCFun
typeFunDef :: Config -> Gamma -> FunDef -> Type
typeFunDef c gam (FunDef rt f dl s) =
  typeParam c gam dl s rt

-- | TCPar
typeParam :: Config -> Gamma -> [Decl] -> [Stmt] -> Type -> Type
typeParam c gam [] s rt = typeStmt c gam s rt
typeParam c gam ((Decl t x):dl) s rt =
  let t' = findInTheta (hat (findInPsi x (psi c))) (theta c)
      gam' = addToGamma x t' gam
  in typeParam c gam' dl s rt

-- Type checking signature for statements.
typeStmt :: Config -> Gamma -> [Stmt] -> Type -> Type

-- | TCDcl
typeStmt c gam ((DeclStmt (Decl t x)):sl) rt =
  let t' = findInTheta (hat (findInPsi x (psi c))) (theta c)
      gam' = addToGamma x t' gam
  in typeStmt c gam' sl rt

-- | TCExp
typeStmt c gam ((ExprStmt e):sl) rt =
  let t = typeExpr c gam e
  in if (isGround t)
     then typeStmt c gam sl rt
     else error $ "expected ground type " ++ show (ppC t)
          ++ " for expression " ++ show e

-- | TCRetZro
typeStmt c gam ((RetStmt (NumLit (IntLit 0))):[]) rt =
  let rt' = (findInTheta (hat rt) (theta c))
  in if isScalar rt'
     then rt'
     else error $ "0 doesn't type with " ++ show (ppC rt') ++ "
        as return"

-- | TCRet
```

```
typeStmt c gam ((RetStmt e):[]) rt =
  let t = typeExpr c gam e
      rt' = (findInTheta (hat rt) (theta c))
  in if isSubTy' t rt'
     then rt'
     else error $ "return doesn't type "
          ++ show (ppC rt) ++ "::" ++ show (ppC rt')

-- Type checking signature for expressions.
typeExpr :: Config -> Gamma -> Expr -> Type

-- | TCLit
typeExpr _ _ (NumLit l) = rho l

-- | TCVar
typeExpr c gam e@(Var x) =
  let t = findInPsi x (psi c)
  in if (t == findInGamma x gam && isGround t)
     then t
     else error $ "Γ and C type mismatch " ++ (show e)

-- | TCFld
typeExpr c gam (FldAcc e x) =
  let pt = typeExpr c gam e
  in case pt of
       PtrTy rt ->
         case findInTheta (hat rt) (theta c) of
           t@(RecTy dl _) -> field x t
           _ -> error $ "expected record in Γ " ++ show (ppC rt)
       _ -> error $ "expected " ++ show (ppC pt) ++ " typed as
          pointer"

-- | TCDrf
typeExpr c gam (Deref e) =
  let t = typeExpr c gam e
  in case t of
    PtrTy t' -> t'
    _ -> error $ "dereference doesn't type check"
```

```haskell
-- | TCAdr
typeExpr c gam (AddrOf e) =
  PtrTy (typeExpr c gam e)

-- | TCAsgZro
typeExpr c gam (BinExpr Assign e1 (NumLit (IntLit 0))) =
  let lht = dropTopQual $ typeExpr c gam e1
  in if isScalar lht
     then lht
     else error $ "0 assignment doesn't type check"

-- | TCAsg
typeExpr c gam (BinExpr Assign e1 e2) =
  let lht = typeExpr c gam e1
      rht = typeExpr c gam e2
  in if isSubTy' rht lht
     then rht
     else error $ "assignment doesn't type check"

-- | TCAdd
typeExpr c gam (BinExpr Add e1 e2) =
  let lht = dropTopQual $ typeExpr c gam e1
      rht = dropTopQual $ typeExpr c gam e2
  in case lht of
    PtrTy _ -> if rht == IntTy
               then lht
               else error $ "expected int as RHS of +"
    _ -> case rht of
      PtrTy _ -> if lht == IntTy
                 then rht
                 else error $ "expected int as LHS of +"
      _ -> if isArith lht && isArith rht
           then highRank lht rht
           else error $ "incompatible types in + (Add)"

-- | TCOr
typeExpr c gam (BinExpr Or e1 e2) =
  let lht = dropTopQual $ typeExpr c gam e1
      rht = dropTopQual $ typeExpr c gam e2
  in if isScalar lht && isScalar rht
     then IntTy
     else error $ "incompatible types in || (OR)"

-- | TCDiv
typeExpr c gam (BinExpr Divide e1 e2) =
  let lht = dropTopQual $ typeExpr c gam e1
      rht = dropTopQual $ typeExpr c gam e2
  in if isArith lht && isArith rht
     then highRank lht rht
     else error $ "incompatible types in / (div)"



-- | Drop top-level qualifier.
dropTopQual :: Type -> Type
dropTopQual (ConstTy t) = dropTopQual t
dropTopQual t = t

-- | Find, in Gamma, the type mapped to an identifier.
findInGamma :: Ident -> Gamma -> Type
findInGamma x gam =
  case Map.lookup x gam of
    Just t -> t
    _ -> error $
         "no τ for identier " ++ show (ppK x) ++ " in Gamma\n"

-- | Add, to Gamma, an identifier and its mapped type.
addToGamma :: Ident -> Type -> Gamma -> Gamma
addToGamma x t gam =
  case Map.lookup x gam of
    Just t -> error $
      show (ppK t) ++ ", indexed by " ++
      show (ppK x) ++ ", is already in Γ\n"
    _ -> Map.insert x t gam
```

A.25 Supporting Functions sc and ari

The implementation of supporting functions for the typing rules, as presented in Figure 3.13.

```
-- | Return whether the type is an arithmetic type.
isArith :: Type -> Bool
isArith IntTy = True
isArith DoubleTy = True
isArith _ = error $ "expected arithmetic type"

-- | Return whether type is an scalar type.
isScalar :: Type -> Bool
isScalar (PtrTy _) = True
isScalar IntTy = True
isScalar DoubleTy = True
isScalar _ = error $ "expected scalar type"

-- | Return the highest ranked of 2 arithmetic types.
highRank :: Type -> Type -> Type
highRank t1 t2 =
  if t1 == IntTy
  then t2
  else t1
```

A.26 The Pretty Printing of the Output

The implementation of our pretty printing of constraints, shapes, types, and the configuration.

```
class PrettyK a where
  ppK :: a -> PP.Doc

instance PrettyK a => PrettyK [a] where
  ppK v = foldr (\x acc -> ppK x PP.<+> PP.text " " PP.<+> acc )
          PP.empty v
```

```
instance PrettyK Ident where
  ppK = PP.text . _x

instance PrettyK TypeId where
  ppK = PP.text . _id

instance PrettyK Type where
  ppK IntTy = PP.text "int"
  ppK DoubleTy = PP.text "double"
  ppK (PtrTy t) = ppK t PP.<> PP.text "*"
  ppK (ConstTy t) = PP.text "const " PP.<> ppK t
  ppK (ArrowTy rt ps) =
    (PP.hcat $ PP.punctuate (PP.text "?  ") (map ppK ps))
    PP.<> PP.text "?  " PP.<> ppK rt
  ppK (RecTy flds x) = PP.text "[record " PP.<> ppK x PP.<>
     PP.char ']'
  ppK (NamedTy x) = ppK x
  ppK (TyVar (Stamp i)) = PP.text "?" PP.<> PP.text (show i)

instance PrettyK Subst where
  ppK Trivial = PP.text "<null-subst>"
  ppK (st :-> t) = ppK st PP.<> PP.text "->" PP.<> ppK t

instance PrettyK K where
  ppK T = PP.text "?"
  ppK B = PP.text "?"
  ppK (k1 :&: k2) = ppK k1 PP.<+> PP.text " ^ " PP.<+> ppK k2
  ppK (Exists t k) =
    PP.text "?" PP.<>
    (foldl (\acc t@(TyVar _) -> acc PP.<> ppK t) PP.empty t)
       PP.<>
    PP.text ". " PP.<> ppK k
  ppK (Def x t k) =
    PP.text "def" PP.<+> ppK x PP.<> PP.colon PP.<> ppK t PP.<+>
    PP.text "in" PP.<+> ppK k
  ppK (Fun f t@(ArrowTy _ _)) =
    PP.text "fun" PP.<+> ppK f PP.<> PP.colon PP.<> ppK t
  ppK (TypeOf x t) =
    PP.text "typeof(" PP.<> ppK x PP.<> PP.text ","
    PP.<> ppK t PP.<> PP.char ')'
  ppK (Syn t1 t2) =
    PP.text "syn " PP.<> ppK t1 PP.<> PP.text " as " PP.<> ppK
       t2
  ppK (Has t fld) =
    PP.text "has" PP.<>
    PP.parens (ppK t PP.<> PP.comma PP.<+>
               ppK (_fx fld) PP.<> PP.colon PP.<>
               ppK (_ft fld))
  ppK (t1 :=: t2) = ppK t1 PP.<> PP.text "?" PP.<> ppK t2
  ppK (t1 :<=: t2) = ppK t1 PP.<> PP.text "?" PP.<> ppK t2

instance PrettyK Stamp where
  ppK (Stamp n) = PP.text "?" PP.<> PP.text (show n)



class PrettyM a where
  ppM :: a -> PP.Doc

instance PrettyM Shape where
  ppM Undefined = PP.text "<undefined>"
  ppM Scalar = PP.text "<scalar>"
  ppM Pointer = PP.text "<pointer>"
  ppM Integral = PP.text "<integral>"
  ppM Floating = PP.text "<floating>"
  ppM Numeric = PP.text "<numeric>"

instance PrettyM M where
  ppM m =
    let
      pp (NumLit v) = PP.text $ show v
      pp (Var x) = PP.text (_x x)
      pp (FldAcc e x) = pp e PP.<> PP.text "->" PP.<> PP.text
         (_x x)
      pp (Deref e) = PP.char '*' PP.<> pp e
      pp (AddrOf e) = PP.char '&' PP.<> pp e
      pp (BinExpr Add e1 e2) = pp e1 PP.<> PP.char '+' PP.<> pp
         e2
      pp (BinExpr Divide e1 e2) = pp e1 PP.<> PP.char '/' PP.<>
         pp e2
      pp (BinExpr Or e1 e2) = pp e1 PP.<> PP.text "||" PP.<> pp
         e2
      pp (BinExpr Assign e1 e2) = pp e1 PP.<> PP.char '=' PP.<>
         pp e2
    in Map.foldrWithKey (\k v acc -> pp k PP.<+> ppM v PP.$$
       acc)
                        PP.empty
                        (_shapes m)



class PrettyAST a where
  fmt :: Int -> a -> PP.Doc

instance PrettyAST a => PrettyAST [a] where
  fmt n (s:sl) =
    fmt n s PP.<> (foldr (\s d -> fmt (n + 1) s PP.<> d)
       PP.empty sl)

instance PrettyAST Expr where
  fmt n e@(NumLit _) = indent n PP.<> PP.text "NumLit"
  fmt n e@(Var _) = indent n PP.<> PP.text "VarDecl"
  fmt n e@(FldAcc x t) = indent n PP.<> PP.text "FieldAccess"
  fmt n e@(Deref e1) = indent n PP.<> PP.text "Deref" PP.<> fmt
     (n + 1) e1
  fmt n e@(AddrOf e1) = indent n PP.<> PP.text "AddrOf" PP.<>
     fmt (n + 1) e1
  fmt n e@(BinExpr _ e1 e2) = indent n PP.<> PP.text "BinExpr"
     PP.<>
                              fmt (n + 1) e1 PP.<> fmt (n + 1)
                                 e2

instance PrettyAST Stmt where
  fmt n (ExprStmt e) = indent n PP.<> PP.text "ExprStmt" PP.<>
     fmt (n + 1) e
  fmt n (DeclStmt _) = indent n PP.<> PP.text "DeclStmt"
  fmt n (RetStmt e) = indent n PP.<> PP.text "RetStmt" PP.<>
     fmt (n + 1) e
```

```
instance PrettyAST Decl where
  fmt n (Decl _ _) = indent n PP.<> PP.text "Decl"

instance PrettyAST FunDef where
  fmt n f@(FunDef _ _ ps s) =
    PP.text "Function" PP.<>
    (foldr (\p d -> fmt (n + 1) p PP.<> d) PP.empty ps) PP.<>
    fmt (n + 1) s

instance PrettyAST Prog where
  fmt n p@(Prog _ fs) =
    (foldr (\f d -> fmt (n + 1) f PP.<> d) PP.empty fs)

indent :: Int -> PP.Doc
indent n = PP.char '\n' PP.<> PP.text (replicate n ' ')


class PrettyC a where
  ppC :: a -> PP.Doc

instance PrettyC TypeId where
  ppC tn = PP.text $ _id tn

instance PrettyC Ident where
  ppC x = PP.text $ _x x

instance PrettyC Decl where
  ppC (Decl t x) = PP.space PP.<>  ppC t PP.<+> ppC x PP.<>
     PP.semi

instance PrettyC Type where
  ppC IntTy = PP.text "int"
  ppC DoubleTy = PP.text "double"
  ppC (PtrTy t) = ppC t PP.<> PP.char '*'
  ppC (ConstTy t) = PP.text "const" PP.<+> ppC t
  ppC (ArrowTy rt pt) =
    ppC rt PP.<> PP.text "(*)" PP.<>
    PP.parens (PP.hcat $ PP.punctuate (PP.text ", ") (map ppC
      pt))
  ppC (RecTy fld x) =
    let prefix = if "struct " `isPrefixOf` (_x x)
                 then ""
                 else "struct "
    in PP.text prefix PP.<> PP.text (_x x) PP.<+>
    PP.braces (PP.hcat $ (map ppC fld))
  ppC (NamedTy x) = PP.text $ _x x
  ppC (TyVar (Stamp n)) = PP.text "?" PP.<> PP.text (show n)


showMap m = show $
  Map.foldrWithKey
  (\k v acc -> PP.lbrace PP.<> PP.space PP.<> ppK k PP.<>
               PP.comma PP.<> PP.space PP.<> ppC v PP.<>
               PP.rbrace PP.<> PP.comma PP.<> PP.space PP.<>
                  acc)
  PP.empty  m

showConfig cfg@(Config { phi, psi, theta, k, kE, kF, kI, kW }) =
  show $
  PP.text (" φ = { " ++ showMap phi) PP.<+> PP.text "}\n" PP.<>
  PP.text (" ψ = { " ++ showMap psi) PP.<+> PP.text "}\n" PP.<>
  PP.text (" Θ = { " ++ showMap theta) PP.<+> PP.text "}\n"
     PP.<>
  PP.text " [Ke] = " PP.<> ppK kE PP.<+> PP.text "\n" PP.<>
  PP.text " [Kf] = " PP.<> ppK kF PP.<+> PP.text "\n" PP.<>
  PP.text " [Ki] = " PP.<> ppK kI PP.<+> PP.text "\n" PP.<>
  PP.text " [Kw] = " PP.<> ppK kW
```

A.27 The μC Parser

The parser we use for μC. This implementation does **not** deal with the ambiguities of the grammar of μA and μB. For a parser that employs the techniques we described in Section 2, please check our tool, PsycheC.

```
langDef = emptyDef {
  Token.identStart = letter,
  Token.identLetter = alphaNum <|> char '_',
  Token.reservedNames =
      [ "int", "double", "const", "return", "struct", "typedef"
         ],
  Token.reservedOpNames =
      [ "*", "/", "+", "||", "=", "&", "->" ]
  }

lexer = Token.makeTokenParser langDef
identifier = Token.identifier lexer
reserved = Token.reserved lexer
reservedOp = Token.reservedOp lexer
parens = Token.parens lexer
braces = Token.braces lexer
integer = Token.integer lexer
float = Token.float lexer
semi = Token.semi lexer
whiteSpace = Token.whiteSpace lexer
comma = Token.comma lexer
symbol = Token.symbol lexer

parseSource :: String -> Either String Prog
parseSource = either (Left . show) Right . parse progParser ""

progParser :: Parser Prog
progParser = Prog <$> many tydefParser <*> many funParser

tydefParser :: Parser TypeDef
tydefParser = TypeDef <$> (reserved "typedef" *>) tyParser <*>
   tyParser <* semi

funParser :: Parser FunDef
funParser = FunDef
            <$> tyParser
            <*> identParser
            <*> parens (declParser `sepBy` comma)
            <*> stmtListParser
```

```
tyParser :: Parser Type
tyParser = f <$> nonPtrTyParser <*> (many starParser)
  where
    f t ts = foldr (\_ ac -> PtrTy ac) t ts

nonPtrTyParser :: Parser Type
nonPtrTyParser = try (qualTyParser intTyParser)
                 <|> try (qualTyParser fpTyParser)
                 <|> try (qualTyParser namedTyParser)
                 <|> qualTyParser recTyParser

starParser :: Parser ()
starParser = () <$ symbol "*"

intTyParser :: Parser Type
intTyParser = IntTy <$ reserved "int"

fpTyParser :: Parser Type
fpTyParser = DoubleTy <$ reserved "double"

namedTyParser :: Parser Type
namedTyParser = f <$> (optionMaybe (reserved "struct")) <*>
   identParser
  where
    f Nothing n = NamedTy n
    f _ n = NamedTy (Ident ("struct " ++ (_x n)))

-- For simplicity, we require in muC that the ‘const’ keyword
   comes before a type,
-- specifier. In standard C, that’s not necessary.
qualTyParser :: Parser Type -> Parser Type
qualTyParser p = f <$> (optionMaybe (reserved "const")) <*> p
  where
    f Nothing t = t
    f _ t = ConstTy t

recTyParser :: Parser Type
recTyParser = RecTy
              <$> braces (declParser ‘sepBy‘ semi)
              <*> identParser

declParser :: Parser Decl
declParser = Decl <$> tyParser <*> identParser

exprParser :: Parser Expr
exprParser = buildExpressionParser table baseExprParser
  where
    table =
      [ [ Prefix (reservedOp "*" >> return Deref) ]
      , [ Prefix (reservedOp "&" >> return AddrOf) ]
      , [ Infix (reservedOp "/" >> return (BinExpr Divide))
         AssocLeft ]
      , [ Infix (reservedOp "+" >> return (BinExpr Add))
         AssocLeft ]
      , [ Infix (reservedOp "||" >> return (BinExpr Or))
         AssocLeft ]
      , [ Infix (reservedOp "=" >> return (BinExpr Assign))
         AssocLeft ] ]

baseExprParser :: Parser Expr
baseExprParser = f <$> fldAccParser <*> (many (reservedOp "->"
   *> identParser))
  where
    f fld xs = foldr (\ x acc -> FldAcc acc x) fld xs

fldAccParser :: Parser Expr
fldAccParser = f <$> primExprParser <*> option id fldAcc
  where
    f x expr = expr x
    fldAcc = flip FldAcc <$> (reservedOp "->" *> identParser)

intParser :: Parser Lit
intParser = IntLit <$> (fromInteger <$> integer)

fpParser :: Parser Lit
fpParser = DoubleLit <$> float

primExprParser :: Parser Expr
primExprParser =  NumLit <$> (try fpParser <|> intParser)
                  <|> Var <$> identParser

stmtParser :: Parser Stmt
stmtParser = RetStmt <$> (reserved "return" *> exprParser)
             <|> try (DeclStmt <$> declParser)
             <|> ExprStmt <$> exprParser

stmtListParser :: Parser [Stmt]
stmtListParser = braces (stmtParser `endBy` semi)

identParser :: Parser Ident
identParser = Ident <$> identifier
```

A.28 The Driver of our Compiler

The driver of μC's compiler (and other supporting code).

```
-- Copyright (c) 2018 Leandro T. C. Melo (ltcmelo@gmail.com)
-- License: GPLv3
-- This implementation focus readability and formalism.

{-# LANGUAGE NamedFieldPuns #-}

import Control.Monad
import Control.Monad.Except
import Control.Monad.State
import Data.List
import Data.Map (Map)
import Data.Set (Set)
import qualified Data.List as List
import qualified Data.Map as Map
import qualified Data.Set as Set
import Debug.Trace
import System.Environment
import System.Exit
import System.IO
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language
import qualified Text.ParserCombinators.Parsec.Token as Token
import qualified Text.PrettyPrint.HughesPJ as PP



main :: IO ()
main = do
  putStrLn "compile muC program"
  args <- getArgs
  case args of
    [file] -> do
      src <- readFile file
      src' <- compile src
      putStrLn $ "\n\n" ++ src'
      let (name, _) = break (== '.') file
      writeFile ("new_" ++ name ++ ".c") src'
    _ -> error "invalid argument"

compile :: String -> IO (String)
compile src =
  case parseSource src  of
    Left err -> return err
    Right p -> do
      debug "AST" (show (fmt 0 p))

      let m = buildLattice p (M $ Map.empty)
      debug "lattice of shapes" (show $ ppM m)

      k <- generateConstraints p m
      debug "K" (show $ ppK k)

      let
        phi_i = Map.empty
        psi_i = Map.empty
        theta_i = (Map.fromList [ (hat IntTy, IntTy),
                                  (hat DoubleTy, DoubleTy) ])
        cfg = Config
```

```
            phi_i
            psi_i
            theta_i
            k
            [] [] [] []
      cfg'@(Config { phi, psi , theta }) <- solveConstraints k
         cfg
      debug "final config" (showConfig cfg')

      let ok = validateSemantics (phi, psi, theta) k
      debug "semantics" (if ok then "OK" else error "does NOT
         hold\n")

      let ts = verifyTyping cfg' Map.empty p
      debug "typing" ("OK")

      let preamble = rewriteInC (theta Map.\\ theta_i)
          src' = preamble ++ src
      return src'


-- | Rewrite inferred types to their C form. This function is
   simplified.
--   Check PyscheC for a complete implementation.
rewriteInC :: Map TypeId Type -> String
rewriteInC theta =
  let
    p (tid@(TypeId id), RecTy _ _ )
      | "struct " `isPrefixOf` id = False
      | otherwise = True
    p (tid@(TypeId id), t)
      | "const " `isPrefixOf` id = False
      | "*" `isSuffixOf` id = False
      | tid == (hat t) = False -- Duplicate: self-definition.
      | otherwise = True
    filtered = filter p (Map.toList theta)

    print (t1, t2) acc = "typedef " ++
                PP.render (ppC t2) ++ " " ++
                PP.render (ppC t1) ++ ";\n" ++ acc

    tydefs = foldr print "" filtered
  in
    (rewriteInC' theta) ++ tydefs

rewriteInC' :: Map TypeId Type -> String
rewriteInC' theta =
  let print' (RecTy _ (Ident x)) = "typedef struct " ++ x ++ "
     " ++ x ++ ";\n"
      print' (PtrTy t) = print' t
      print' (ConstTy t) = print' t
      print' _ = ""
      print (tid@(TypeId id), t) acc
        | "struct " `isPrefixOf` id = PP.render (ppC t) ++ ";\n"
        | otherwise = print' t ++ acc
  in foldr print "" (Map.toList theta)
```