UNIVERSIDADE DO MINHO

Extenções de Alto Nível para uma Linguagem de Programação Criptográfica

Vicente Machado Fernandes

Escola de Engenharia
Departamento de Informtica

27 de outubro de 2010

Declaração

Nome: Vicente Machado Fernandes

Endereço eletrónico: vinc13e@gmail.com

Telefone: 914970639

Número do Bilhete de Identidade: 12599891

Título da Tese: Extenções de alto nível para uma linguagem de programação criptográfica

Orientador: Professor Doutor José Carlos Bacelar Almeida

Ano de conclusão: 2010

Designação do Mestrado: Mestrado em Informática

Universidade do Minho, _____

Assinatura: _____

UNIVERSIDADE DO MINHO

Resumo

Escola de Engenharia
Departamento de Informtica

Vicente Machado Fernandes

A implementação de técnicas criptográficas numa linguagem de programação comum nem sempre é fácil, pois estas técnicas requerem muitas vezes operações complexas como exponenciações, operações sobre grupos, corpos ou outras estruturas algébricas que nem sempre estão disponíveis nestas linguagens. Para resolver este problema existem linguagens vocacionadas para a criptografia que dispõe de um alargado conjunto de recursos para simplificar a implementação de técnicas criptográficas.

Um desses casos é a linguagem de programação CAO que está a ser desenvolvida no âmbito do Projecto FP7 CACE[1] (Computer Aided Cryptography Engineering). Esta linguagem foi desenhada com o objectivo de facilitar a implementação de algoritmos criptográficos numa notação que se aproxime daquela utilizada nas publicações científicas e normas criptográficas, e será suportada por um compilador que automatizará algumas optimizações e deverá garantir a segurança do código implementado.

Assim, neste projecto de dissertação vamos estudar e explorar algumas extensões de alto nível para a linguagem CAO. Estas extensões são baseadas num mecanismo de qualificação de tipos — uma forma simples e eficaz de estabelecer restrições sobre os tipos de uma linguagem. Cada qualificador é visto como uma propriedade atómica que qualifica os tipos gerais. Com estes qualificadores é possível estabelecer uma noção de expressão e variável constante em CAO, realizar uma análise relativa ao *information flow* dos programas ou aumentar a confiabilidade do *software* que implementa primitivas criptográficas.

[1] http://www.cace-project.eu/

UNIVERSIDADE DO MINHO

Abstract

Escola de Engenharia
Departamento de Informtica

Vicente Machado Fernandes

The implementation of cryptographic techniques on a common programming language is not always easy, because these techniques require complex operations such as exponentiation, operations on groups, fields or other algebraic structures that are not always available in these languages. To solve this problem there are languages aimed at cryptography that offers a broad range of features to simplify the implementation of cryptographic techniques.

One of these cases is the the programming language CAO that is being developed under the project FP7 CACE[2] (Computer Aided Cryptography Engineering). This language was designed with the aim of facilitating the implementation of cryptographic algorithms in a notation that came close to those used in scientific publications and cryptographic standards, and will be supported by a compiler that will automate some optimizations and should ensure the security of the implemented code.

This dissertation project seeks to explore some high-level extensions for the CAO language. These extensions are based on a mechanism of type qualifiers — a practical and useful mechanism for specifying properties that are not captured by the usual type systems. A type qualifier can be seen as an atomic property that qualifies the general types. With these qualifiers is possible to establish a notion of constant variable and constant expression in CAO, prevent undesirable information flows of programs or increase the reliability of the software that implements cryptographic primitives.

[2] http://www.cace-project.eu/

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As aplicações de *software* são cada vez mais parte integrante do nosso quotidiano. Grande parte das operações fundamentais que regem a nossa sociedade dependem de sistemas informáticos que são cada vez mais poderosos e complexos. No entanto, este aumento na complexidade dos sistemas desenvolvidos leva também a um aumento na responsabilidade de quem os desenvolve. Assim, é fundamental que este aumento na complexidade seja acompanhado por um aumento da nossa capacidade de assegurar a sua qualidade. Vulnerabilidades em sistemas informáticos podem originar ataques com elevados custos financeiros e podem, em casos extremos onde é utilizado *software* critico, levar a falhas com consequências ainda mais nefastas como aconteceu, por exemplo, com o sistema de gestão de ambulâncias em Londres no ano de 1992 [1] onde uma falha na componente responsável pela recepção de chamadas de emergência originou 46 mortes por falta de uma assistência atempada. Por estas razões, ao logo dos últimos anos, tem aumentado o interesse e o investimento em áreas que lidam directamente com a segurança e a confiabilidade dos sistemas de *software*.

Paral tal existem diversas técnicas como testes [2], *design by contract* [3] verificação de modelos de *software* [4] entre outras, que de forma isolada ou combinada entre si resolvem com graus de eficácia e dificuldade diferente algumas das vulnerabilidades que os sistemas de *software* apresentam.

Paralelamente, com o avanço das tecnologias da informação, a questão da segurança nas transmissões de dados fez aumentar em muito importância da criptografia. O seu impacto na esfera cultural, social e económica é cada vez maior, o que obriga a que as aplicações criptográficas sejam cada vez mais complexas

e robustas. Assim, para construir tais aplicações são necessárias linguagens de programação sensíveis à problemática da criptografia moderna. Estas linguagens deverão disponibilizar aos seus utilizadores um alargado conjunto de mecanismos que facilitem a sua usabilidade e a segurança do próprio código.

Por outro lado, é desejável que existam técnicas que permitam verificar formalmente as propriedades de segurança de programas escritos nessas linguagens, garantindo assim a consistência do código fonte. Algo que, em código criptográfico, tem uma importância acrescida dado este se encontra associado a funções críticas ao nível da segurança das aplicações.

Assim, é fundamental que uma linguagem de programação criptográfica reúna estas duas condições: ajudar o programador na implementação de técnicas criptográficas disponibilizando para o efeito primitivas adequadas; e fornecer mecanismos intrínsecos à linguagem que permitam a verificação formal dos programas nela escritos.

1.1 Projecto *Cace*

O projecto FP7 CACE (Computer Aided Cryptography Engineering) tem como objectivo desenvolver e implementar um alargado conjunto de ferramentas destinadas à especificação e desenvolvimento de aplicações no domínio do *software* criptográfico. O desenvolvimento de *software* criptográfico tem uma dificuldade acrescida uma vez que a segurança e a confiabilidade de muitas aplicações modernas são garantidas através de técnicas criptográficas de elevada complexidade. No entanto, para confiarmos numa aplicação criptografia não basta termos a garantia que as técnicas que esta implementa são consistentes. Temos também que confiar nas ferramentas nas quais estas foram implementadas, pois muitas ataques a técnicas criptográficas exploram vulnerabilidades que possam existir na implementação e não na técnica em si (como por exemplo ataques baseados em *buffer overflow* [5]).

Linguagem CAO. Ao desenvolver software criptográfico de alta performance o programador é confrontado com uma vasta gama de problemas. Estes problemas relacionados com a segurança de software são normalmente mais difíceis de diagnosticar e resolver do que os problemas funcionais comuns. Normalmente

```
def f(n : int) : int, int {
   def a, b : int;
   a, b := 0, 1;
   while (n > 0) {
      a, b := b, a + b;
      n := n - 1;
   }
   return a, b;
}
```

FIGURA 1.1: Definição da função de Fibonacci em CAO.

os problemas funcionais originam erros que fazem com que os programas parem de executar enquanto os problemas de segurança tipicamente só são detectados quando alguém explora essas vulnerabilidades. Além disso, os alvos dos ataques a software criptográfico são escolhidos deliberadamente pelos atacantes. Estes procuram operações cuja segurança não tenha sigo correctamente garantida pelo programador para desta forma atingir os seus objectivos.

O CAO é uma linguagem de programação de alto nível destinada à produção de código criptográfico que tem como objectivo evitar muitos destes problemas relacionados com a segurança do software. Naturalmente, o CAO pode ser usado para implementar outro tipo de primitivas que não criptográficas. Um exemplo disso é a função, responsável por calcular os elementos da sequência de Fibonacci, que podemos ver na Figura 1.1.

No Capítulo 2 vamos apresentar de forma detalhada as caracteristicas do CAO e a sua especificação formal.

Do projecto CACE resultam três ferramentas fundamentais:

- **Interpretador de CAO.** Permite a implementação de primitivas criptográficas (e não só) em CAO e a sua respectiva avaliação.
- **CAO-SL.** Linguagem de especificação baseada em anotações para programas escritos na linguagem CAO. Estas anotações são incorporadas nos comentários do código e são ignorados pelo compilador de CAO. A linguagem é inspirada em ACSL [6] e tal como esta pretende lidar com propriedades relacionadas com o comportamento dos programas.
  Com anotações CAO-SL é possível, por exemplo, definir restrições sobre

funções Cao onde são identificados de forma inequívoca quais as funcionalidades e garantias que a função prevê e quais os pré-requisitos que devem ser atendidos na obtenção de tais garantias.

- **Compilador CALF.** A linguagem CALF, como vamos ver detalhadamente no Capítulo 2, é uma versão enriquecida da linguagem CAO que inclui algumas características adicionais relativamente a esta. A linguagem CALF tem dois objectivos principais: fornecer um ambiente de programação mais amigável de forma a ajudar o programador na implementação das primitivas criptográficas permitindo por exemplo reduzir a probabilidade de erros de programação, permitir a reutilização, gestão de código etc; Tirando partido das características adicionais que a linguagem dispõe relativamente ao CAO, permite produzir um código mais confiável que possa ser convertido para CAO funcionando assim o CALF como verificador e pré-processador de código para CAO. Esta tradução segue um conjunto de transformações automáticas e deverá produzir um código CAO padrão que preserva as propriedades desejáveis que as extensões do CALF nos oferecem.

Todas as ferramentas deselvolvidas no projecto CACE foram implementadas em Haskell[1].

1.2 Sistemas de Tipos

Sistemas de tipos [7] desempenham um papel fundamental nas linguagens de programação actuais. Estes fornecem uma forma natural e intuitiva de expressar e verificar a estrutura dos programas permitindo que a presença de alguns erros seja detectada automaticamente.

Propriedades expectáveis de um sistema de tipos. Os tipos geralmente utilizados em linguagens de programação têm características intrínsecas que os diferenciam de outros tipos de anotações de programas. Normalmente, as anotações sobre o comportamento dos programas são baseadas em comentários informais com a especificação pretendida que têm que ser verificadas por ferramentas externas à própria linguagem. Por oposição, os tipos são intrínsecos à linguagem,

[1] http://www.haskell.org/haskellwiki/Introduction

sendo um programa imediatamente rejeitado se violar a disciplina de tipos pré-estabelecida. Genericamente, é comum considerar-se as seguintes características associadas aos sistemas de tipos:

- Os sistemas de tipos devem ser decidíveis, ie, deve existir um algoritmo (*typechecker*) que garante que um programa é bem comportado. Além disso, os sistemas de tipos devem ter a capacidade de detectar erros de execução antes que eles aconteçam.
- Os sistemas de tipos devem ser transparentes, um programador deve ser capaz de prever quando um programa vai passar no *typechecker* ou não. Quando a verificação de tipos falha, as razões que levaram a tal ocorrência devem ser facilmente identificaveis.

Em qualquer linguagem o sistema de tipos é fundamental para garantir o correcto funcionamento do software que nela venha a ser implementado. Também neste trabalho o sistema de tipos desempenha um papel fundamental. As funcionalidades estudadas na dissertação e que apresentaremos ao longo do documento têm como base alterações ao sistema de tipos de CAO.

1.3 Objectivos da Dissertação

Este trabalho de dissertação tem como objectivo principal estudar e implementar um sistema que permita qualificar os tipos da linguagem CAO.

Várias linguagens de programação dispõem de qualificadores de tipos, quer de forma nativa quer em bibliotecas externas à linguagem. Normalmente os qualificadores são usados para de alguma forma refinar os tipos da linguagem. Cada qualificador tem a si associado uma propriedade que tem que ser respeitada por todas as variáveis que venham a ser declaradas com o tipo sobre o qual este qualificador está definido. Um exemplo de qualificador usado massivamente é o `const` disponível em linguagens como C ou C++ [8]. Este qualificador é usado para garantir que determinadas variáveis só possam ser instanciadas uma única vez. Algo semelhante acontece no *Java* [9] com o qualificar `final`. Ainda em C/C++ bibliotecas de OpenGL® [10] dispõem de diversos qualificadores. Entre os quais

estão, por exemplo, os qualificadores `highp` ou `lowp`, que actuam sobre os tipos `int` e `float` e obrigam as variáveis a ter uma determinada precisão.

Além destes qualificadores tradicionais que vêm predefinidos nas linguagens algumas ferramentas permitem que os utilizadores definam os seus próprios qualificadores. Exemplo disso é a *framework CQual* [11] desenvolvida por Jeffrey Foster que permite que se adicionam qualificadores à linguagem C.

No CAO optamos por uma situação mista onde existem qualificadores predefinidos e onde é possível definir novos qualificadores. Os predefinidos são usados em duas situações distintas. Para raciocinar sobre expressões e variáveis constantes e para garantir diversas propriedades relacionadas com a não interferência [12]. Os qualificadores definidos pelo utilizador tem um alcance bastante maior e podem ser utilizados nas mais variadas situações, mas em particular o sistema de qualificadores do CAO está desenhado de forma a permitir que os programadores definam qualificadores que possam ser usados na implementação de primitivas criptográficas e desta forma contribuir para o aumento da segurança do software criptográfico.

1.4 Organização da Dissertação

O resto desta dissertação está organizado da seguinte forma.

No Capítulo 2 apresentamos a linguagem CAO, a sua especificação formal e a sua versão enriquecida (linguagem CALF).

O Capítulo 3 é dedicado aos qualificadores de tipos e está dividido em duas partes. Na primeira parte são apresentadas as principais vantagens de um sistema baseado na qualificação de tipos. Além disso são apresentadas algumas das noções fundamentais necessárias na construção de tal sistema. A segunda parte é dedicada aos qualificadores definidos pelo utilizador. Inicialmente é apresentada a sintaxe formal que suporta a definição de novos qualificadores. Em seguida explicamos a sua semântica e as diferentes formas de definição de qualificadores. Além disso, apresentamos o sistema de tipos da linguagem CAO com qualificadores, o algoritmo responsável pela inferência destes qualificadores e as suas características fundamentais. Finalmente, são referidos e comparados alguns dos trabalhos mais relevantes existentes na literatura sobre mecanismos de qualificação de tipos.

Nos Capítulos 4 e 5 apresentamos respectivamente os qualificadores predefinidos `const` e `untrusted`. O primeiro usado para estabelecer uma noção de constante sobre as variáveis e expressões do CAO e o segundo para garantir algumas propriedades relativas à não interferência.

No Capítulo 6 concluímos e apresentamos algumas funcionalidades, que não tendo sido objecto de estudo nesta dissertação, se apresenta como possíveis desenvolvimentos futuros.

Capítulo 2

Especificação da Linguagem CAO/CALF

2.1 Descrição da Linguagem

Neste capítulo vamos apresentar a especificação formal da linguagem CAO e a forma como esta é estendida para assim dar origem à linguagem CALF. Para uma melhor familiarização com a linguagem ao longo do capítulo vamos apresentar diversos pequenos excertos de código CAO.

Tipos e variáveis. Os programas em CAO utilizam variáveis para representar os dados que são manipulados pelos operadores. Nestas variáveis é sempre possível identificar o seu nome e o seu tipo.

Os tipos do CAO, como na maioria das linguagens comuns, podem ser de duas variantes: **tipos primitivos** ou **tipos sinónimos**. Os tipos primitivos descrevem os objectos mais básicos de um programa. No CAO, como podemos ver na Tabela 2.1, existem 5 tipos primitivos.

Os tipos sinónimos são tipos mais complexos construídos a partir dos tipos primitivos. Estes tipos são construídos com a palavra reservada `typedef` como podemos ver no seguinte exemplo:

Void	Tipo vazio
Int	Inteiros de precisão arbitrária
Bool	Booleanos
Bits$[i]$	Strings de Bits de tamanho i
Mod m	Anel ou corpo definido em m
Vector$[i]$ of α	Vector com i, elementos do tipo α
Matrix$[i,j]$ of α	Matriz com $i \times j$, elementos do tipo $\alpha \in \mathcal{A}$

$$\mathcal{A} = \{\mathsf{Int}, \mathsf{Mod}\ m, \mathsf{Matrix}[i,j]\ \mathsf{of}\ \beta \mid \beta \in \mathcal{A}\}$$

Tabela 2.1: Tipos primitivos do CAO

```
typedef point := struct [
        def x : int;
        def y : int;
        def z : int;
        ];
```

Declaração de variáveis. A declaração de variáveis em CAO é análoga à da maioria das linguagens de programação imperativas. Em cada declaração de variável é utiliza a palavra reservada `def` (que também é usada na declaração de funções e de tipos sinónimos) seguida do identificador da variável e do respectivo tipo associado. É também possível declarar várias variáveis numa única instrução, para tal é apenas necessário que estas variáveis tenham todas o mesmo tipo. No seguinte exemplo podemos ver a declaração de três variáveis com o tipo `int` em CAO.

```
def x, y, z: int;
```

Expressões. As expressões são formadas através de variáveis e literais que são combinadas com recurso a operadores pré-definidos. No CAO estes operadores podem ser unários e binários e são divididos entre aritméticos (soma, subtracção, multiplicação, divisão, redução modular e raiz quadrada), *bit-wise* (NOT, AND, XOR, OR, *shift*, *rotate* e *concat*) e lógicos (NOT, AND, XOR, OR, e os operadores de comparação ==, ! =, <, >, <= e >=).

Atribuições. A atribuição de valor a uma variável é uma instrução fundamental em qualquer linguagem de programação. A ideia é que possamos avaliar qualquer *RValue* e copiar o resultado para um *LValue*, ou seja, atribuir o valor do lado direito para o lado esquerdo. As atribuições no CAO são idênticas às do C, com a diferença de no C as atribuições serem expressões enquanto que no CAO cada atribuição é um comando. Exemplo de atribuição em CAO:

```
def x: int;
x := y + z;
```

Além disso, o CAO permite atribuições paralelas, ou seja, várias atribuições numa única instrução. Nestes casos, todos os *RValue* são avaliados no pré-estado do comando.

```
def x, y, z : int;
x, y, z := a, b, a+b;
```

Selecção e concatenação. Em linguagens como o C ou *Java* o operador de acesso a um índice de um vector (ou matriz) é usado para seleccionar um determinado elemento desse vector. Para tal usamos a notação $v[i]$ para seleccionar o i-ésimo elemento do vector v.
No CAO, esta operação pode ser utilizada em diferentes contextos além da simples selecção de um elemento. Um desses casos é a selecção de intervalos de valores (*ranges*) onde é possível seleccionar um subconjunto contínuo de elementos de um vector (ou matriz) e copia-los para um segundo vector, tal é sempre possível desde que o tipo e o tamanho deste segundo vector sejam compatíveis com o subconjunto seleccionado. A sintaxe desta operação é a seguinte:

```
def v : vector[12] of int;
def v' : vector[6] of int;
v' := [0..5];
```

Um Segundo caso é a operação de concatenação que permite que dados dois vectores se determine um terceiro vector que contém todos os elementos dos dois

primeiros onde a ordem pela qual estes foram passados ao operador é preservada. Esta operação é generalizável para mais de dois vectores, como podemos ver no seguinte exemplo:

```
def x : unsigned bits[8];
def y : unsigned bits[6];
y := x[0..2] @ x[3..5] @ x[6..8];
```

Estruturas de controlo e iteração. O CAO utiliza algumas das estruturas de controlo mais comuns em diversas linguagens de programação de uso generalizado. Uma delas é o `if` que é o usado da mesma forma que no C. Esta operação faz um teste de uma condição booleana e consoante o resultado deste teste executa ou não um conjunto de instruções. Existe ainda a estrutura `if/else` cuja única diferença relativamente ao `if` é a execução se um segundo bloco de instruções caso o teste booleano avalie em falso. Contrariamente ao que acontece no C no CAO apenas expressões booleanas podem ser usadas como condição no teste.

Outra estrutura de controlo presente no CAO é o `while`, onde mais uma vez, a sintaxe e semântica coincidem com a do C. Esta operação é constituida por uma condição booleana e um bloco de intruções que são executadas enquanto a condição for válida.

Além dos ciclos convencionais como o `while` o CAO fornece dois replicadores o `seq` e o `seq by`. Estes replicadores são constituídos por uma variável constante do tipo `int` (definida pelo replicador) e por dois índices que limitam o início e o fim da replicação. No caso do `seq by` existe um terceiro valor que indica o "salto" que é dado ente cada iteração e a seguinte. Finalmente, em cada replicador existe um bloco de instruções que são executadas em cada iteração. O seguinte exemplo mostra um replicador `seq by` entre 0 e 10 com espaçamento 2.

```
seq i := 0 to 10 by 2 {
        v[i] := v[i] + 1;
}
```

Funções e procedimentos. As funções em CAO são definidas de forma semelhante ao C, cada declaração especifica o nome da função, o tipo de retorno, os argumentos e os seus respectivos tipos e por fim o corpo da função. Uma das principais diferenças relativamente à definição de funções em C está relacionada com a semântica dos argumentos já que em CAO estes são sempre passados por valor e não é possivel simular passagens por referência como acontece no C. Outra diferença considerável relativamente ao C é o facto do em CAO as funções permitirem o retorno de múltiplos valores.

2.2 Sintaxe

Como vimos, informalmente, na secção anterior a sintaxe do CAO é semelhante à de outras linguagens imperativas de uso generalizado como o C. Na Figura 2.1 apresetamos a respectiva sintaxe formal. No entanto, para simplificar a apresentação das extensões adicionadas ao CAO, ao longo desta dissertação utilizamos apenas um subconjunto da linguagem. Assim, a sintaxe que apresentamos na Figura 2.1 corresponde apenas a esse subconjunto do CAO. A sintaxe completa da linguagem pode ser consultada no *deliverable* 5.3 [13] do projeto CACE.

Esta sintaxe será posteriormente enriquecida de forma a suportar algumas das extensões consideradas nesta tese.

2.3 Regras de Tipagem

Tal como acontece na maioria das linguagens de programação as regras de sintaxe do CAO não são suficientemente rigorosas para impedir que programas com erros de tipagem sejam aceites. Desta forma, existe uma fase inicial na interpretação de programas em CAO que tem como objectivo verificar se os programas após serem analisados sintacticamente de acordo com as regras de sintaxe apresentadas na secção anterior violam alguma das regras da disciplina de tipos imposta, evitando assim que programas inválidos possam ser avaliados. Além disso, esta fase é responsável pela recolha de informação sobre o tipo associado às expressões dos programas. Posteriormente esta informação é usada para facilitar a forma como a semântica operacional da linguagem é expressada.

Domínios de Expressão Sintática

i, j, k, n	:	**Num**	Literais de inteiros
L	:	**Lit**	Literais
x	:	$\mathbf{Id}_V$	Identificadores de variáveis
f	:	$\mathbf{Id}_F$	Identificadores de funções
fi	:	$\mathbf{Id}_{SF}$	Identificadores de campos de `structs`
sid	:	$\mathbf{Id}_S$	Identificadores de `structs`
tid	:	$\mathbf{Id}_T$	Identificadores de tipos
e	:	**Exp**	Expressões
c	:	**Stm**	*Statements*
dv	:	$\mathbf{Dec}_V$	Declaração de variáveis
df	:	$\mathbf{Dec}_F$	Declaração de funções
dt	:	$\mathbf{Dec}_T$	Declaração de tipos
l	:	**Lv**	*LValues*
τ	:	**Types**	Tipos
pg	:	**Progs**	Programas

Regras de Sintaxe

$$
\begin{array}{rcl}
e & ::= & L \mid x \mid -e \mid e_1{+}e_2 \mid e_1{-}e_2 \mid e_1{*}e_2 \mid e_1 \,/\, e_2 \mid e_1 \,\%\, e_2 \mid e_1{**}\, e_2 \mid e.fi \mid \\
 & & f(e_1, \ldots, e_n) \mid e_1 == e_2 \mid e_1 \,!= \, e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 <= e_2 \mid \\
 & & e_1 >= e_2 \mid e_1 \| e_2 \mid e_1 \,\&\&\, e_2 \mid e_1[e_2] \mid e_1[i..j] \mid e_1 \,@\, e_2 \\
 & & \\
l & ::= & x \mid l[e] \mid l[i..j] \mid l.fi \\
 & & \\
c & ::= & dv \mid l_1, \ldots, l_i := e_1, \ldots, e_j; \mid \mathsf{return}\ e_1, \ldots, e_n; \mid f(e_1, \ldots, e_n); \mid \\
 & & \mathsf{while}\ (e)\ \{\ c_1; \ldots; c_n\ \} \mid \mathsf{if}\ (e)\ \{\ c_1; \ldots; c_n\ \} \mid \\
 & & \mathsf{if}\ (e)\ \{\ c_{11}; \ldots; c_{1n}\ \}\ \mathsf{else}\ \{\ c_{21}; \ldots; c_{2n}\ \} \\
 & & \\
dt & ::= & \mathsf{typedef}\ tid := \tau; \mid \mathsf{typedef}\ sid := \mathsf{struct}\ [\ \mathsf{def}\ fi_1 : \tau_1; \ldots; \mathsf{def}\ fi_n : \tau_n;\ ]; \\
 & & \\
dv & ::= & \mathsf{def}\ x : \tau; \mid \mathsf{def}\ x_1, \ldots, x_n : \tau; \mid \mathsf{def}\ x : \tau := e; \mid \mathsf{def}\ x : \tau := \{\ e_1, \ldots, e_n\ \}; \\
 & & \\
df & ::= & \mathsf{def}\ f\ (x_1 : \tau_1, \ldots, x_n : \tau_n) : rt\ \{\ c_1; \ldots; c_n\ \} \\
 & & \\
rt & ::= & \mathsf{void} \mid \tau_1, \ldots, \tau_n \\
 & & \\
\tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{vector}\ [k]\ \mathsf{of}\ \tau \mid tid \mid sid \\
 & & \\
pg & ::= & dv \mid dt \mid df \mid pg_1\ pg_2
\end{array}
$$

FIGURA 2.1: Sintaxe formal do CAO.

Como referimos anteriormente, ao longo desta trabalho vamos apenas utilizar um subconjunto da linguagem CAO para evidenciar a utilização as extensões adicionadas à linguagem. Além disso, como estas extensões, nomeadamente os qualificadores de tipos, têm como base alterações ao sistema de tipos da linguagem apresentaremos, ao longo da dissertação, apenas o sistema de tipos relativo a esse subconjunto do CAO já considerando essas extensões.

Nesta secção vamos apresentar as regras de tipagem responsáveis pelas coerções do CAO. As restantes regras, relativas às construções da linguagem serão apresentadas no próximo capítulo após ser feito o necessário enquadramento dos qualificadores de tipos com o sistema de tipos do CAO.

Coerção de tipos. A coerção entre tipos é usada para coagir valores de um determinado tipo para um segundo tipo sem que para tal seja necessário definir explicitamente um *cast* entre estes dois tipos. Este mecanismo de coerção entre tipos deve ser capaz de garantir a inexistência de perda de informação durante o processo.

Para suportar coerções entre tipos no CAO é necessário previamente definir uma relação de coerção $\leq$ entre tipos coercíveis. Esta relação é uma ordem parcial, portanto é simultaneamente reflexiva, transitiva e anti-simétrica.

$$\frac{\vdash_{eq} \tau_1 = \tau_2}{\vdash_{\leq} \tau_1 \leq \tau_2}$$

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2 \quad \vdash_{\leq} \tau_2 \leq \tau_3}{\vdash_{\leq} \tau_1 \leq \tau_3}$$

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2 \quad \vdash_{\leq} \tau_2 \leq \tau_1}{\vdash_{eq} \tau_1 = \tau_2}$$

No CAO é possível realizar coerções apenas entre os tipos `bits` (`signed` e `unsigned`) e `int`, entre vectores, matrizes e entre polinómios.

2.4 Semântica de Avaliação

Como as extensões ao CAO estudadas nesta dissertação são em grosso modo transparentes à semântica da linguagem não iremos aqui apresentar exaustivamente as construções semânticas do CAO.

A semântica de avaliação do CAO pode ser consultada no *deliverable* 5.3 [13] do projeto CACE.

2.5 CALF

Nesta dissertação nenhuma das extensões estudadas será adicionada à linguagem CALF. Todo o trabalho incide sobre o CAO. No entanto, o CALF tem um conjunto de situações onde estas extensões (nomeadamente os qualificadores de tipos) seriam uma mais-valia para a linguagem. Desta forma, apresentamos nesta secção uma breve descrição desta linguagem.

Como referimos no capítulo introdutório a linguagem CALF é uma versão enriquecida da linguagem CAO que inclui extensões de alto nível. Esta linguagem tem dois objectivos principais: fornecer um ambiente de programação mais amigável de forma a ajudar o programador na implementação das primitivas criptográficas permitindo uma melhor gestão e reutilização de código e uma diminuição da probabilidade de ocorrência de erros de programação; e, tirando partido das extensões que a linguagem dispõe relativamente ao CAO produzir um código mais confiável que possa ser convertido para CAO funcionando assim o CALF como verificador e pré-processados de código. Esta tradução segue um conjunto de transformações automáticas e deverá produzir um código CAO padrão que preserva as propriedades desejáveis que as extensões do CALF nos oferecem.

2.5.1 Sintaxe

A sintaxe do CALF, que podemos ver na Figura 2.2, é estendida a partir da sintaxe do CAO mostrada na Figura 2.1. A principal diferença desta sintaxe estendida está relacionada com a notação para os operadores de ordem superior [] e <> que não existem no CAO. Além disso, em diversas situações, a sintaxe do CALF tem

uma notação que permite que se expressem parâmetros nos tipos, ou seja, permite que se utilizem tipos dependentes em determinadas construções, como na definição de funções ou de tipos sinónimos.

$$
\begin{array}{rcl}
e & ::= \ldots \mid & f <e_1, \ldots, e_n> (e_{n+1}, \ldots, e_m) \mid e_1[f]e_2 \mid [f]e \mid \langle f \rangle e \\
c & ::= \ldots \mid & f <e_1, \ldots, e_n> (e_{n+1}, \ldots, e_m) \\
df & ::= \ldots \mid & \mathsf{def}\ f <x_1 : \tau_1, \ldots, x_n : \tau_n> (x_{n+1} : \tau_{n+1}, \ldots, x_m : \tau_m) : \\
 & & rt\ \{\ c_1, \ldots, c_l\ \} \\
dt & ::= \ldots \mid & \mathsf{typedef}\ tid <x_1 : \tau_1, \ldots, x_n : \tau_n> := \tau \mid \\
 & & \mathsf{typedef}\ sid <x_1 : \tau_1, \ldots, x_n : \tau_n> \\
 & & := \mathsf{struct}\ [\ \mathsf{def}\ fi_{n+1} : \tau_{n+1}; \ldots; \mathsf{def}\ fi_m : \tau_m\ ] \\
\tau & ::= \ldots \mid & tid <e_1, \ldots, e_n> \mid sid <e_1, \ldots, e_n>
\end{array}
$$

FIGURA 2.2: Sintaxe extendida do CALF.

2.5.2 Extensões

As principais características adicionais que se pretendem para o CALF são as seguintes.

- **Operadores de ordem superior.** Uma das principais características da linguagem é suportar funções de ordem superior que permitem redefinir de forma simplificada alguns dos operadores sobre estruturas de dados como vectores e matrizes. Entre estes operadores estão o *zip-with* que recebe como parâmetro duas expressões com o mesmo tipo e tamanho e uma função e devolve uma terceira expressão com o mesmo tamanho e tipo das anteriores e que resulta da aplicação da função pelos sub-expressões das expressões recebidas. Ou o *map-parttern* que recebe uma expressão e uma função como argumento e devolve uma expressão com o mesmo tipo e tamanho da primeira que resulta da aplicação da função recebida a cada uma das suas sub-expressões.

- **Polimorfismo.** Permitir funções polimórficas que são funções genéricas que podem ser utilizadas com argumentos de diferentes tipos e que são úteis, por exemplo, para manipular vectores de dimensão desconhecida e inteiros modulares cujo módulo é desconhecido. Este mecanismo tem uma sintaxe semelhante ao dos *templates* de algumas linguagens de programação comuns.

- **Tradução para CAO.** Outra das características fundamentais da linguagem CALF está relacionada com o facto de ser possível traduzir o código

CALF para CAO. Funcionando assim o CALF como pré-processador de código para CAO, onde cada programa pode ser visto como um template genérico para programas CAO que podem ser instanciados com parâmetros diferentes.

Além disso, a ferramenta de compilação de CALF permite que os programadores escrevam um código mais robusto que pode ser traduzido para código CAO com a garantia este satisfaz o conjunto de propriedades assumidas pelo compilador de CAO.

Capítulo 3

Qualificadores de Tipos

3.1 Introdução

Como vimos no Capítulo 1 os sistemas de tipos das linguagens de programação desempenham um papel fundamental na especificação e verificação de propriedades desejáveis dos programas. No entanto, a maioria das linguagens fornecem um conjunto muito limitado de mecanismos para expressar essas propriedades. Isto é razoável uma vez que é impossível antecipar as propriedades que os programadores pretendem especificar bem como as diferentes formas de verificar essas propriedades.

Nesse sentido, a utilização de qualificadores de tipos mostra ser um mecanismo extremamente útil na especificação de propriedades que não são capturadas pelos sistemas de tipos comuns. Cada qualificador pode ser visto como uma propriedade atómica que qualifica os tipos gerais. Assim, podemos utilizar esta técnica em alternativa a outro tipo de técnicas que têm como objectivo definir restrições sobre as estruturas de dados utilizadas em diversas linguagens de programação, tipicamente estas restrições são definidas através de invariantes e escritas na forma de anotações [14] no código fonte. Naturalmente, a utilização de qualificadores não substitui totalmente a utilização de ferramentas de verificação baseadas em anotações uma vez que muitas das propriedades normalmente capturadas por estas ferramentas não podem ser garantidas pelos qualificadores em tempo de compilação.

Neste trabalho de dissertação os qualificadores de tipos são utilizados para adicionar informação aos tipos gerais do CAO. Esta informação é adicionada sob a forma de propriedades atómicas que qualificam os tipos. Desta forma ao qualificarmos um determinado tipo estamos a garantir que todas as variáveis declaradas com este tipo qualificado cumprem com a especificação definida pelo qualificador.

Neste capítulo vamos apresentar algumas das noções fundamentais necessárias na construção de uma ferramenta de qualificação de tipos e mostrar o mecanismo que permite adicionar novos qualificadores de tipos à linguagem CAO.

3.2 Subtipagem – Noções Básicas

Resumidamente, um tipo τ_1 é subtipo de um segundo tipo τ_2 ($\tau_1 <: \tau_2$) se todos os valores que têm o tipo τ_1 são ainda valores com o tipo τ_2. Por exemplo, na matemática todos os números naturais são números inteiros e todos os números inteiros são reais. Assim, é perfeitamente natural que linguagens de programação tenham, por exemplo, o tipo `Nat` (que representa os números naturais) como subtipo de `Int` (que representa os números inteiros). O mesmo acontece com os inteiros relativamente aos números reais.

Outra forma onde facilmente se pode verificar a utilização da noção de subtipo é através de *records*. Por exemplo, se considerarmos um *record* com os campos `age` e `name` e assumindo que o campo `name` é do tipo `string` e o campo `age` tem o tipo `int`, podemos dizer que o *record* $\{age : \texttt{int},\ name : \texttt{string}\}$ é subtipo do *record* $\{age : \texttt{int}\}$. Em algumas linguagens, como O'Haskell [15], a noção de subtipagem pode ainda ser usada na definição de tipos algébricos. Neste caso, o próprio mecanismo de definição de tipos permite que estes sejam definidos como subtipo de outro tipo.

Apesar de esta noção de subtipagem ser relativamente simples e intuitiva, em determinadas situações pode não ser possível identificar um tipo como sendo subtipo de um segundo tipo. Para resolver este problema muitos sistemas de tipos usam funções de coerção que transformam os elementos do subtipo em elemento do supertipo.

Num sistema de qualificadores de tipos como o do CAO a subtipagem é usada fundamentalmente quando se pretende realizar coerções ou *casts* entre um dado

tipo τ e o tipo resultante da qulificação de τ com um qualificador q_{id}, ou seja, *casts* ou coerções entre τ e $q_{id}\ \tau$. Nestes casos, a função de coerção usada para converter elementos do tipo qualificado no tipo sem o qualificador é sempre a função identidade. Isto porque, como veremos mais adiante, para qualquer tipo τ e qualificador q_{id}, $q_{id}\ \tau$ é subtipo de τ ($q_{id}\ \tau <: \tau$).

3.3 Ordens Parciais e Reticulados

Numa ferramenta que disponibilize um conjunto de qualificadores de tipos a uma linguagem é de extrema utilidade relacionar de alguma forma os qualificadores entre si. Em particular, se existir uma relação de ordem parcial entre os qualificadores podemos, por exemplo, permitir coerções entre tipos qualificados.

Nesta secção vamos apresentar uma breve introdução à teoria que suporta as ordens parciais e descrever algumas das propriedades básicas dos reticulados [16] obtidos a partir destas relações de ordem.

definição 3.1. Um conjunto parcialmente ordenado é um par $\mathcal{P} = (P, \leq)$ onde P é um connjunto não vazio e $\leq$ é uma relação binária em P que satisfaz, para qualquer $x, y, z \in P$, as seguintes propriedades:

1. $x \leq x$ (reflexividade)
2. se $x \leq y$ e $y \leq x$ então $x = y$ (antisimetria)
3. se $x \leq y$ e $y \leq z$ então $x \leq z$ (transitividade)

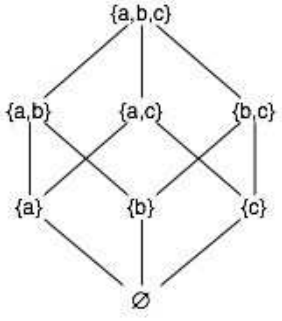

FIGURA 3.1: Conjunto potência de $\{a, b, c\}$ ordenado pela relação $\subseteq$.

Na figure 3.1 podemos ver um exemplo clássico de uma ordem parcial formada pelos elementos do conjunto $\mathcal{P}\{S\}$ ordenados pela relação $\subseteq$ e onde $S \neq \emptyset$. Como esta ordem é parcial existem dois ou mais elementos incomparáveis (por exemplo $\{a\}$ and $\{c\}$ na figura 3.1). Analogamente, quando não existem elementos incomparáveis no conjunto ordenado, este diz-se totalmente ordenado.

Duas propriedades extremamente úteis das relações de ordem são dadas pela noção de ínfimo e de supremo.

definição 3.2. Seja $(A,\ \leq)$ uma ordem parcial e seja $X \subset A$.

- Chama-se **limitante inferior** de X em A a todo o elemento $a \in A$ tal que $a \leq x$ para todo $x \in X$.
- Chama-se **limitante superior** de X em A a todo o elemento $a \in A$ tal que $x \leq a$ para todo $x \in X$.
- Chama-se **ínfimo** de X em A ao maior dos limitantes inferiores de X em A. Este elemento é denotado por $\bigwedge X$. Para quaisquer dois elementos $x,\ y\ \in X$ o ínfimo $\{x,\ y\}$ denota-se $x \wedge y$.
- Chama-se **supremo** de X em A ao menor dos limitantes superiores de X em A. Este elemento é denotado por $\bigvee X$. Para quaisquer dois elementos $x,\ y\ \in X$ o supremo $\{x,\ y\}$ denota-se $x \vee y$.

Finalmente, com uma ordem parcial e com as noções de ínfimo e supremo define-se um reticulado da seguinte forma:

definição 3.3. Seja $(A,\ \leq)$ uma ordem parcial, se para quaisquer par de elementos $x,\ y\ \in A$ existem $x \wedge y$ e $x \vee y$ então $(A,\ \leq)$ é um **reticulado**.

3.4 Exemplo Introdutório

Num contexto criptográfico as restrições que os qualificadores impõem sobre os tipos são extremamente úteis em diversas situações. Vejamos o seguinte exemplo da definição de um corpo primo em CAO presente na figura 3.2. Os corpos primos [17] são usados massivamente na implementação de primitivas criptográficas, nomeadamente em técnicas de criptografia assimétrica [18] nas quais o módulo

```
def p : int := 103;
typedef Zp := mod[p];
```

FIGURA 3.2: Definição da extensão de corpo.

do corpo deverá ser um número primo, caso contrário a correcção da técnica é facilmente comprometida.

O exemplo da figura 3.2 é constituído por uma variável (`p`) do tipo inteiro que representa o modulo do corpo e pelo tipo `Zp` usado para representar os seus elementos. A variável `p` é do tipo `int` e portanto pode tomar qualquer valor, primo ou não. Isto acontece porque o sistema de tipos do CAO, tal como a maioria dos sistemas de tipos tradicionais, não têm qualquer mecanismo que garanta que um determinado número é ou não primo.

Em algumas linguagens é possível utilizar testes de primalidade para esse fim, no entanto, qualquer teste de primalidade tem uma elevada complexidade temporal o que torna a sua utilização pouco exequível. Além disso, mesmo com a garantia que um determinado número é primo ao longo das várias operações que envolvem a implementação de uma primitiva criptográfica é muito fácil que a informação que garante que uma determinada variável tem a si associado um número primo se perca. É aqui que os qualificadores de tipos se tornam uma mais-valia. Com um qualificador `prime` que qualifique os inteiros é possivel definir variáveis do tipo `prime int` obrigando assim a que qualquer valor associado a estas variáveis seja um "número primo". Na prática o qualificador não nos garante que um número é primo, esta garantia terá que ser dada sempre pelo programador. O qualificador garante apenas que a partir do momento em que o programador define uma variável como sendo `prime int` então esta só poderá tomar valores que este admita serem de facto primos, mesmo que tal não aconteça.

No exemplo apresentado na figura 3.2 o qualificador `prime` pode ser usado na declaração da variável `p`. Assim qualquer valor que esta possa tomar será um número primo e portanto, as variáveis com o tipo `Zp` serão sempre inteiros modulo `p` ou seja, são elementos do corpo $\mathbb{Z}_p$.

3.5 Qualificadores Definidos pelo Utilizador

Os qualificadores de tipos têm com objectivo estabelecer um conjunto de propriedade que actuam sobre os tipos da linguagem. Estas propriedades podem ser vistas como invariantes que têm que ser respeitados por todas as variáveis declaradas com estes tipos.

Muitas linguagens de programação disponibilizam de forma nativa alguns qualificadores para utilizações concretas. São exemplo disso o qualificador `const` do C/C++ ou o qualificador `final` do *Java*. No entanto, este conjunto limitado de qualificadores que estas linguagens têm não é, em muitos casos, suficiente para que os programadores possam definir as propriedades desejáveis sobre os tipos. Por esta razão é de extrema utilidade permitir que estes definam os seus próprios qualificadores e desta formam estabeleçam um conjunto de propriedades sobre os tipos que se adeqúe ao código que desenvolvem.

Como veremos em detalhe ao longo do resto do capítulo, no CAO introduzimos duas formas distintas de definir novos qualificadores, através de uma cláusula `assume`, ou através de cláusulas `case`. Para simplificar a escrita, ao logo do restante deste documento, designaremos os qualificadores definidos com cada uma destas cláusulas como qualificadores `assume` e qualificadores `case`, respectivamente.

3.5.1 Sintaxe

Para dar suporte à definição de novos qualificadores no CAO é necessário criar uma sintaxe própria que permita embeber esta definição nos programas CAO.

Esta sintaxe, que podemos ver na Figura 3.3, utiliza um subconjunto da sintaxe do CAO e além disso, dispõe de palavras reservadas que permitem diferenciar as diferentes formas de definir os qualificadores. Nomeadamente a palavras `case` e `assume`, que como veremos em detalhe mais à frente, introduzem duas formas diferentes de definir novos qualificadores.

Domínios de Expressão Sintática

i, j, k, n	:	**Num**	Literais de inteiros
b_{lit}	:	**Bool**	Literais de booleanos
x	:	$\mathbf{Id}_V$	Identificadores de variáveis
q_{id}	:	$\mathbf{Id}_Q$	Identificadores de qualificadores
e	:	**Exp**	Expressões
c	:	$\mathbf{Stm}_Q$	*Statements* dos qualificadores
cb	:	$\mathbf{Stm}_Q$	*Statements* dos blocos `case`
mv	:	$\mathbf{Dec}_V$	Declaração de meta-variáveis
pd	:	$\mathbf{Stm}_Q$	*Statements* dos padrões usados nas regras de tipagem
sc	:	$\mathbf{Stm}_Q$	*Statements* das condições laterais
dq	:	$\mathbf{Dec}_Q$	Declaração de qualificadores
τ	:	**Types**	Tipos

Regras de Sintaxe

dq ::= qualifier $q_{id}(x : \tau)$ { c }

c ::= assume(b_{lit}) | case x of cb

b_{lit} ::= true | false

cb ::= $mv\ tr$ (where sc) | $mv\ pd$ (where sc) cb

mv ::= def $x_1, \ldots, x_n : \tau_1, \ldots, \tau_n$

pd ::= $x \mid lit\ x \mid x_1 + x_2 \mid x_1 - x_2 \mid x_1 * x_2 \mid x_1 \,/\, x_2 \mid x_1 \,\%\, x_2$
$\mid x_1 ** x_2 \mid -x_1$

sc ::= $e_1 == e_2 \mid e_1 != e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 \leq e_2 \mid$
$sc \,\&\&\, sc \mid sc \,\|\, sc \mid !sc \mid q_{id} < x > \mid b_{lit}$

e ::= $L \mid x \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \,/\, e_2 \mid e_1 \,\%\, e_2 \mid e_1 ** e_2 \mid e.fi \mid (\tau)\ e \mid$
$\mid e_1[e_2] \mid e_1[i..j] \mid e_1 \,@\, e_2$

FIGURA 3.3: Sintaxe formal dos qualificadores de tipos da linguagem CAO.

A partir do momento em que é possível no CAO definir qualificadores de tipos é necessário alterar a sintaxe da declaração de tipos, variáveis e funções de forma a permitir o uso de qualificadores. Todas as restantes construções sintáticas do CAO permanecem inalteradas. Desta forma, a sintaxe da definição de variáveis, tipos e funções com qualificadores em está representada na Figura 3.4, onde os símbolos $\mathcal{Q}_i$ representam conjuntos de qualificadores.

$$
\begin{array}{rcl}
dt & ::= & \mathsf{typedef}\ sid := \mathsf{struct}\ [\ \mathsf{def}\ fi_1 : \mathcal{Q}_1\ \tau_1; \ldots; \mathsf{def}\ fi_n : \mathcal{Q}_n\ \tau_n;\ ]; \\
 & & \mid \mathsf{typedef}\ tid := \mathcal{Q}\ \tau; \\
dv & ::= & \mathsf{def}\ x : \mathcal{Q}\ \tau; \mid \mathsf{def}\ x_1, \ldots, x_n : \mathcal{Q}\ \tau; \mid \mathsf{def}\ x : \mathcal{Q}\ \tau := e; \\
 & & \mid \mathsf{def}\ x : \mathcal{Q}\ \tau := \{\ e_1, \ldots, e_n\ \}; \\
df & ::= & \mathsf{def}\ f\ (x_1 : \mathcal{Q}_1\ \tau_1, \ldots, x_n : \mathcal{Q}_n\ \tau_n) : rt\ \{\ c_1; \ldots; c_n\ \} \\
rt & ::= & \mathsf{void} \mid \mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n
\end{array}
$$

FIGURA 3.4: Sintaxe alterada do CAO com qualificadores de tipos.

3.5.2 Semântica

Na linguagem CAO cada definição de um novo qualificador para além de indicar o tipo que vai qualificar permite definir as regras de tipagem do qualificador, possibilitando assim, por exemplo, que se restrinja a sua utilização a apenas um subconjunto de expressões que tenham esse tipo. Como podemos ver na Figura 3.5 a primeira linha de cada definição de novo qualificador indica o seu identificador e o tipo sobre o qual este vai ser utilizado. Em seguida são definidas as regras de tipagem. Estas podem ser de dois tipos, e são definidas, respectivamente, através da palavra `assume` e `case`.

Cláusula assume. É usada em situações em que o compilador não consegue, num período de tempo razoável, garantir a consistência desta nova definição, nestes casos a responsabilidade é assumida explicitamente pelo programador. Um exemplo desta utilização é o qualificador `prime` defindo no segundo exemplo da Figura 3.5. Como vimos na Secção 3.4 este qualificador é extremamente útil num contexto criptográfico, uma vez que pode ser usado para informar tanto o compilador como o programador que um determinado valor expectável é um número primo. Algo que é crucial para garantir a correcção de diversas técnicas criptográficas como, por exemplo, as técnicas baseadas em corpos finitos.

```
1   qualifier positive (x: int){
2           case x of
3             def A: int;
4             lit A, where A > 0;
5             def A, B: int;
6             A + B, where positive<A> && positive<B>;
7             def A, B: int;
8             A * B, where positive<A> && positive<B>;
9   }
10
11  qualifier prime (x: int){
12          assume(true);
13  }
```

FIGURA 3.5: Definição dos qualificadores `positive` e `prime`.

Cláusula case. É usada quando se pretende definir explicitamente as regras de tipagem do novo qualificador. Cada cláusula `case` representa uma regra de tipagem e é constituída por um padrão, uma condição e um conjunto de meta-variáveis que serão usadas no resto da regra.

As meta-variáveis definem-se da mesma forma que as variáveis do CAO e são válidas apenas na regra em que são definidas. Portanto a mesma variável pode ser redefinida em várias regras no mesmo qualificador.

Cada padrão é constituído por um literal ou por uma expressão onde pode ser utilizado apenas um operador, unário ou binário e cujos operandos são as meta-variáveis definidas anteriormente ou literais.

Aqui é necessário ter em conta o seguinte, quando pretender-mos mais do que uma utilização para uma determinada expressão esta deve ser garantida na condição lateral (através da disjunção das varias restrições pretendidas) e não na duplicação do padrão por várias regras.

Vejamos o seguinte exemplo:

```
qualifier q(x : int){
        case x of
          def X, Y :int;
          lit X + lit Y, where X > 0;          1
          def X, Y : int;
          lit X + lit Y, where Y > 0;          2
```

```
        }

        qualifier q(x : int){
                case x of
                  def X, Y: int;
                  lit X + lit Y, where X > 0 || Y > 0; 3
        }
```

Se pretendermos que na definição de um qualquer qualificador exista o padrão $X + Y$ com a restrição $X > 0$ e esse mesmo padrão com a restrição $Y > 0$ então esse padrão deve ser definido numa regra, como vemos no exemplo 3, cuja condição lateral é a disjunção das duas restrições que pretendemos definir. Se utilizarmos duas regras com o mesmo padrão, casos 1 e 2, o segundo será sempre ignorado uma vez que a função que verifica se uma dada expressão faz "match" com os padrões utilizados na definição dos qualificadores pára quando encontra um padrão que combine com a expressão e a pesquisa é sequencial e preserva a ordem pela qual as regras de tipagem são definidas.

As condições laterais são expressões booleanas onde podem ocorrer literais, as meta-variáveis definidas nessa regra e a invocação de qualificadores incluindo o qualificador que está a ser definido. Como veremos mais à frente, a quando da utilização de variáveis qualificadas com cláusulas `case` sempre que uma expressão faz "match" com um determinado padrão a condição lateral associada a esse padrão é avaliada sendo as meta-variáveis que nela ocorrem instanciadas pelos valores associados à expressão que está a ser utilizada.

A primeira cláusula `case` na Figura 3.5 indica que cada literal do tipo inteiro tem a si associado o qualificador `positive` desde que o seu valor atómico seja maior que zero. A segunda cláusula `case` garante que a soma de duas expressões do tipo `positive int` é ainda uma expressão do tipo `positive int`. Analogamente, a última cláusula `case` indica que o produto entre duas expressões `positive int` é também uma expressão `positive int`.

3.5.3 Subtipagem

Como referimos na Secção 3.2, a noção de subtipagem é extremamente útil em diversos contextos, nomeadamente quando se pretende realizar *casts* ou coerções

entre os diversos tipos de uma linguagem.

Ao implementarmos um sistema de qualificação de tipos estamos também a estabelecer uma relação de subtipagem entres estes tipos. Assim, dado $\mathcal{Q}$ o conjunto de qualificadores do CAO e q_{id} elemento de $\mathcal{Q}$ são válidas as seguintes regras que definem a relação de subtipagem entres os tipos qualificados do CAO.

$$\frac{}{q_{id}\ \tau <: \tau}\ (ax)$$

$$\frac{\tau_1 <: \tau_2}{q_{id}\ \tau_1 <: q_{id}\ \tau_2}\ (<:)$$

FIGURA 3.6: Regras de subtipagem dos tipos qualficados em CAO.

A primeira regra (ax) na Figura 3.6 indica que para qualquer tipo τ, se este for qualificado com o qualificador q_{id} então o tipo qualificado é subtipo de τ. A segunda regra $(<:)$ garante que se o tipo τ_1 é subtipo de τ_2 então, para qualquer qualificador q_{id}, $q_{id}\ \tau_1 <: q_{id}\ \tau_2$.

3.5.3.1 Relação de Qualificação

Além da definição de novos qualificadores a nosso sistema de qualificação permite que se estabeleça uma relação de ordem parcial entre os vários qualificadores. Desta forma, sempre que um utilizador define um novo qualificador pode, se assim o pretender, adicionar esse qualificador a essa ordem de qualificação. Por exemplo, se pretendermos que o tipo `prime int` seja subtipo de `odd int`, basta indicar esta informação na definição do qualificador `prime`, como podemos ver no exemplo seguinte, para tal é necessário que o qualificador `odd` tenha sido definido previamente. Nestas situações dizemos que `prime` é menor que `odd` na relação de qualificação e escrevemos `prime` $\leq$ `odd`.

```
qualifier prime (x: odd int){
        assume(true);
}
```

Esta relação entre os qualificadores é de extrema utilidade porque permite que se estenda a relação de subtipagem do CAO apresentada na Figura 3.6. Desta forma,

seja $(\mathcal{Q}, \leq)$ a relação de ordem entres os qualificadores do CAO a seguinte regra (que garante que para quaisquer dois qualificadores q_1 e q_2 se $q_1 \leq q_2$ na relação dos qualificadores então, para qualquer tipo τ, $q_1\ \tau$ é subtipo de $q_2\ \tau$) é ainda válida.

$$\frac{q_1 \leq q_2}{q_1\ \tau <: q_2\ \tau}\ (\leq)$$

3.5.3.2 *Casts* e Coerções

Coerções. Tal como foi referido anteriormente, o estabelecimento de uma relação de subtipagem entre os qualificadores permite que se identifique de forma rigorosa as situações em que é possível realizar coerções entre os tipos qualificados no CAO. Isto acontece porque esta relação de subtipagem pode ser usada para definir uma relação de coerção. No caso do CAO a relação de subtipagem é utilizada para estender a relação de coerção já existente. Assim, se um tipo τ_1 é subtipo de outro tipo τ_2 é sempre possivel realizar uma coerção do supertipo (τ_2) para o subtipo (τ_1). Esta propriedade é capturada pela seguinte regra.

$$\frac{\tau_1\ <:\ \tau_2}{\vdash_{\leq}\ \tau_1 \leq\ \tau_2}$$

Com esta regra, e com a regra de subtipagem (ax) apresentada na Figura 3.6 podemos, em particular, concluir que um tipo qualificado é coercível para o seu supertipo. Assim sendo, podemos considerar a seguinte regra que estende a relação de coerção de tipos apresentada na Secção 2.3.

$$\frac{}{\vdash_{\leq}\ qid\ \tau \leq\ \tau}$$

Onde q_{id} é o identificador de um qualificador. Na Secção 3.7 vamos mostrar alguns exemplos de código CAO onde são efectuadas coerções envolvendo tipos qualificados.

Casts. Ao contrário das coerções, a utilização de *casts* entre diferentes tipos tem especificidades que tornam esta operação delicada. Intuitivamente, quando fazemos um *cast* do tipo τ_1 para o tipo τ_2 estamos a perder a informação vinculada pelo tipo τ_1 para eventualmente ficarmos com a informação vinculada pelo segundo tipo.

Vejamos o exemplo da lingagem C. Se considerarmos o tipo `int` como subtipo de `float`,

$$\texttt{int} <: \texttt{float}$$

representados graficamente da seguinte forma.

Neste caso passagens do nível inferior para o superior são coerções e portanto são efectuadas implicitamente. Pelo contrário, ao efectuarmos uma passagem do nível superior para o inferior estamos a perder a precisão adicional que o tipo `float` oferece relativamente ao tipo `int`, por essa razão no C esta conversão é permitida apenas com recurso a um *cast*.

```
x: int; y: float;
y = 3.14;
x = (int) y;
```

Algo similar acontece, por exemplo, na conversão entre classes e sub-classes das linguagens orientadas a objectos.

No sistema de qualificação do CAO este mecanismo de *cast* mantém-se, no entanto a sua utilização tem particularidades que o diferenciam do exemplo anterior. Seja q_{id} um qualificador e τ um tipo, como vimos anteriormente, $q_{id}\ \tau$ é subtipo de τ,

$$q_{id}\ \tau <: \tau$$

graficamente, o nível superior é τ (supertipo) e o nível inferior é $q_{id}\ \tau$ (subtipo).

Aqui a passagem do nível inferior para o superior é implícita porque $q_{id}\ \tau$ é subtipo de τ e como vimos na Secção 3.5.3.2, existe uma coerção de $q_{id}\ \tau$ para τ. Por outro lado, e tal como no exemplo do C, a passagem do nível superior para o inferior é possível apenas através de um *cast*. Mas, como o tipo inferior é qualificado, a propriedade que o qualificador q_{id} nos oferece pode não ser garantida pela instância do tipo superior aquando do *cast*. Assim, a utilização de um *cast* de um tipo qualificado para o seu supertipo pode levar facilmente à ocorrência de inconsistências no código desenvolvido. Desta forma, cabe ao programador identificar as situações em que é aceitável realizar estes *casts*.

O objectivo destes *casts* para tipos qualificados passa precisamente por cobrir situações em que apesar de o programador ter a garantia que uma determinada variável cumpre com a propriedade vinculada por um dado qualificador, a definição desse qualificador não é suficientemente expressiva para garantir tal propriedade a essa variável, e portanto o *cast* funciona nestes casos como uma "bênção" dada pelo programador. Outra situação onde estes casts são frequentes está relacionada com a inicialização das variáveis qualificadas com um qualificador `assume`. Nestes casos a sua inicialização terá que passar sempre por um *cast* ou por um *assignement* envolvendo uma segunda variável com o mesmo qualificador.

3.6 Sistema de Tipos

Nesta secção vamos apresentar o sistema de tipos da linguagem CAO com qualificadores. Como a linguagem que utilizamos para formalizar o sistema de qualificação é um subconjunto da linguagem CAO as regras de tipagem que apresentamos correspondem apenas a esse subconjunto do CAO. Alám disso, as coerções existentes no CAO são omitidas nesta representação. Assim, quando nos referimos a um tipo τ, este representa qualquer τ' desde que $\tau' \leq \tau$.

A apresentação das regras de tipagem segue um estilo tipicamente usado para este tipo de representação. A principal diferença relativamente às apresentações

comuns está relacionada com os símbolos $\mathcal{Q}_i$ que neste contexto representam conjuntos de qualificadores de tipos. E com a função *fQual* que aqui é usada para determinar o conjunto de qualificadores que é obtido por cada regra. Esta função poderia ser substituída por instâncias concretas de cada qualificador, no entanto, optamos por esta apresentação para evitar a necessidade de apresentar a mesma regra para cada qualificador. Desta forma a apresentação está mais compacta e customizável.

O símbolo Γ é usado para representar o ambiente que contém as declarações de tipos, variáveis e funções. Para as variáveis este ambiente está dividido em duas partes Γ_G e Γ_L, que contêm respectivamente as varáveis globais e variáveis locais. Para simplificar a notação utilizamos isoladamente o simbolo Γ quando as declarações em questão tanto podem estar em Γ_G como em Γ_L.

Antes de passarmos à apresentação das regras é importante realçar que, tal como vimos na sintaxe do CAO, o parâmetro que define o tamanho dos vectores tem que obrigatoriamente ser um literal inteiro. Também nos *ranges* de vectores apenas literais podem ser usados.

Assim, define-se a função ϕ_l que dado qualquer literal inteiro devolve o seu respectivo valor associado,

$$\phi_l(n) = n$$

esta função será usada nas regras responsáveis pela tipagem da definição e *ranges* de vectores.

3.6.1 Regras de Tipagem

Literais.

$$\frac{}{\Gamma \vdash \mathsf{true} :: \mathcal{Q}\ \mathsf{Bool}}$$

$$\frac{}{\Gamma \vdash \mathsf{false} :: \mathcal{Q}\ \mathsf{Bool}}$$

$$\frac{}{\Gamma \vdash i :: \mathcal{Q}\ \mathsf{Int}}\ i \in \mathbb{Z}$$

Variáveis, chamadas de funções e projecções de estruturas.

$$\frac{\Gamma(x) = \mathcal{Q}\ \tau}{\Gamma \vdash x :: (fQual\ \mathcal{Q})\ \tau} \quad x \in dom(\Gamma)$$

$$\frac{\Gamma_G(f) = (\mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n) \to \mathcal{Q}\ \tau \quad \Gamma \vdash e_1 :: \mathcal{Q}_1\ \tau_1 \quad \ldots \quad \Gamma \vdash e_n :: \mathcal{Q}_n\ \tau_n}{\Gamma \vdash f(e_1, \ldots, e_n) :: (fQual\ \mathcal{Q})\ \tau}$$

onde, $f \in dom(\Gamma)$

$$\frac{\Gamma_G(fi) = \mathcal{Q}_1\ \tau_1 \to \mathcal{Q}_2\ \tau_2 \quad \Gamma \vdash e :: \mathcal{Q}_1\ \tau_1}{\Gamma \vdash e.fi :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau_2} \quad fi \in dom(\Gamma)$$

Operadores aritméticos.

$$\frac{\Gamma_G \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \mathsf{Int}} \quad \oplus \in \{+, -, *, /, **, \%\}$$

$$\frac{\Gamma \vdash e :: \mathcal{Q}\ \tau}{\Gamma \vdash -e :: (fQual\ \mathcal{Q})\ \tau}$$

Operadores booleanos.

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \tau \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \tau}{\Gamma \vdash e_1 \oplus e_2 :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \mathsf{Bool}} \quad \oplus \in \{==, !=\}$$

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \mathcal{Q}_1\ \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \mathsf{Bool}} \quad \oplus \in \{<, \leq, >, \geq\}$$

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Bool} \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \mathsf{Bool}}{\Gamma \vdash e_1 \oplus e_2 :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \mathsf{Bool}} \quad \oplus \in \{||, \&\&, \hat{}\hat{}\}$$

$$\frac{\Gamma \vdash e \leq \mathcal{Q}\ \mathsf{Bool}}{\Gamma \vdash !e :: (fQual\ \mathcal{Q})\ \mathsf{Bool}}$$

Operadores sobre vectores.

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Vector}[i]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \mathsf{Vector}[j]\ \mathsf{of}\ \tau}{\Gamma \vdash e_1 \mathbin{@} e_2 :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \mathsf{Vector}[i+j]\ \mathsf{of}\ \tau}$$

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Vector}[i]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash e_1[e_2] :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau}$$

$$\frac{\Gamma \vdash e :: \mathcal{Q}_1\ \mathsf{Vector}[k]\ \mathsf{of}\ \tau \quad \Gamma \vdash i :: \mathsf{Int} \quad \Gamma \vdash j :: \mathsf{Int} \quad \phi_l(i) = i \quad \phi_l(j) = j}{\Gamma \vdash e[i..j] :: (fQual\ \mathcal{Q}_1)\ \mathsf{Vector}[j-i+1]\ \mathsf{of}\ \tau}$$

onde, $k > j, j \geq i \geq 0$

Casts.

$$\frac{\Gamma \vdash e :: \mathcal{Q}_1\ \tau}{\Gamma \vdash (\mathcal{Q}_2\ \tau)\ e :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau}$$

LValues.

$$\frac{\Gamma(x) = \mathcal{Q}\ \tau}{\Gamma \vdash x :: (fQual\ \mathcal{Q})\ \tau} \quad \tau \in dom(\Gamma)$$

$$\frac{\Gamma_G(fi) = \mathcal{Q}_1\ \tau_1 \to \mathcal{Q}_2\ \tau_2 \quad \Gamma \vdash l :: \mathcal{Q}_1\ \tau_1}{\Gamma \vdash l.fi :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau_2} \quad fi \in dom(\Gamma)$$

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \mathsf{Vector}[i]\ \mathsf{of}\ \tau \quad \Gamma \vdash e :: \mathcal{Q}_2\ \mathsf{Int}}{\Gamma, \vdash l[e] :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau}$$

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \mathsf{Vector}[k]\ \mathsf{of}\ \tau \quad \Gamma \vdash i :: \mathcal{Q}_2\ \mathsf{Int} \quad \Gamma \vdash j :: \mathcal{Q}_3\ \mathsf{Int} \quad \phi_l(i) = i \quad \phi_l(j) = j}{\Gamma \vdash l[i..j] :: (fQual\ \mathcal{Q}_1)\ \mathsf{Vector}[j-i+1]\ \mathsf{of}\ \tau}$$

onde, $k > j, j \geq i \geq 0$

Statements. Para as regras de tipagem dos *statements* é necessário introduzir uma notação adicional. Assim, o símbolo ρ representa o tipo de retorno da função na qual o *statement* está definido. O símbolo $\bullet$ representa o possível tipo de retorno da função e tem como objectivo distinguir os casos em que o bloco executa explicitamente o *statement* `return` dos casos em que este *statement* é executado dentro do bloco.

Ainda relativamente aos *statements*, para simplificar a sua apresentação, as regras da definição, *assignment* e *return* paralelo são omitidas, isto porque estas regras são generalizações das correspondentes regras de definição, *assignment* e *return* singular.

Declarações.

$$\frac{}{\Gamma \models_{\rho} \mathsf{def}\ x : \mathcal{Q}\ \tau :: (\bullet, \Gamma[x :: \mathcal{Q}\ \tau])} \quad x \notin dom(\Gamma)$$

$$\frac{}{\Gamma \models \mathsf{typedef}\ tid\ :=\ \mathcal{Q}\ \tau :: (\bullet, \Gamma)}$$

$$\frac{\Gamma, \epsilon[x_1 :: \mathcal{Q}_1\ \tau_1, \ldots, x_n :: \mathcal{Q}_n\ \tau_n] \models_{\tau} c :: (\mathcal{Q}\ \tau, \Gamma'_G)}{\Gamma, \epsilon \models \mathsf{def}\ fp(x_1 : \mathcal{Q}_1\ \tau_1, \ldots, x_n : \mathcal{Q}_n\ \tau_n) : \mathcal{Q}'\ \tau'\ \{c\} :: (\bullet, \Gamma[fp :: ((\mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n) \to \mathcal{Q}'\ \tau')])}$$

onde, $\tau' \neq \mathsf{void}$ e $fp \notin dom(\Gamma)$

$$\frac{\Gamma, \epsilon[x_1 :: \mathcal{Q}_1\ \tau_1, \ldots, x_n :: \mathcal{Q}_n\ \tau_n] \models_{()} c; \mathsf{return}() :: ((), \Gamma'_G)}{\Gamma, \epsilon \models \mathsf{def}\ fp(x_1 : \mathcal{Q}_1\ \tau_1, \ldots, x_n : \mathcal{Q}_n\ \tau_n) : \mathsf{void}\ \{c\} :: (\bullet, \Gamma[fp :: ((\mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n) \to ())])}$$

onde, $fp \notin dom(\Gamma)$

$$\frac{\epsilon, \epsilon, \epsilon \models d_1 :: (\bullet, \Gamma_{G_1}) \quad \ldots \quad \Gamma_{G_{n-1}}, \epsilon, \epsilon \models d_n :: (\bullet, \Gamma_G)}{\epsilon, \epsilon, \epsilon \models d_1; \ldots; d_n :: (\bullet, \Gamma_G)} \quad \mathsf{main} :: () \to () \in \Gamma$$

Declaração e inicialização de variáveis.

$$\frac{\Gamma \vdash e :: \mathcal{Q}_1\ \tau}{\Gamma \models_{\rho} \mathsf{def}\ x : \mathcal{Q}_2\ \tau := e :: (\bullet, \Gamma[x :: \mathcal{Q}_2\ \tau])} \quad x \notin dom(\Gamma)$$

onde, $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \tau \ldots, \Gamma \vdash e_n :: \mathcal{Q}_n\ \tau}{\Gamma \models_\rho \mathsf{def}\ x : \mathcal{Q}\ \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau := \{e_1, \ldots, e_n\} :: (\bullet, \Gamma[x :: \mathcal{Q}\ \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau])}$$

onde, $x \notin dom(\Gamma)$

Assignements.

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \tau \qquad \Gamma \vdash e :: \mathcal{Q}_2\ \tau}{\Gamma \models_\tau l := e :: (\bullet, \Gamma[l :: \mathcal{Q}_1\ \tau])}$$

onde, $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$

Funções.

$$\frac{\Gamma(fp) = ((\mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n) \to \mathcal{Q}\ \tau) \quad \Gamma \vdash l :: \mathcal{Q}'\ \tau \quad \Gamma \vdash e_1 :: \mathcal{Q}_1'\ \tau_1 \ldots \Gamma \vdash e_n :: \mathcal{Q}_n'\ \tau_n}{\Gamma \models_\tau l := fp(e_1, \ldots, e_n) :: (\bullet, \Gamma)}$$

onde, $fp \in dom(\Gamma)$, $\mathcal{Q}' \subseteq \mathcal{Q}$ e $\mathcal{Q}_i \subseteq \mathcal{Q}_i'$ com, $1 \leq i \leq n$.

$$\frac{\Gamma(fp) = ((\mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n) \to ()) \quad \Gamma \vdash e_1 :: \mathcal{Q}_1'\ \tau_1 \ldots \Gamma \vdash e_n :: \mathcal{Q}_n'\ \tau_n}{\Gamma \models_\tau fp(e_1, \ldots, e_n) :: (\bullet, \Gamma)}$$

onde, $fp \in dom(\Gamma)$ e $\mathcal{Q}_i \subseteq \mathcal{Q}_i'$ com, $1 \leq i \leq n$

$$\frac{\Gamma \vdash e :: \mathcal{Q}\ \tau}{\Gamma \models_\tau \mathsf{return}\ e :: (\mathcal{Q}\ \tau, \Gamma)}$$

Sequenciação.

$$\frac{\Gamma \models_\tau c_1 :: (\bullet, \Gamma') \quad \Gamma' \models_\tau c_2; \ldots; c_n :: (\rho, \Gamma'')}{\Gamma \models_\tau c_1; \ldots; c_n :: (\rho, \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

$$\frac{\Gamma \models_\tau c_1 :: (\mathcal{Q}\ \tau, \Gamma') \quad \Gamma' \models_\tau c_2; \ldots; c_n :: (\rho, \Gamma'')}{\Gamma \models_\tau c_1; \ldots; c_n :: (\mathcal{Q}\ \tau, \Gamma'')} \quad \rho \in \{\mathcal{Q}\ \tau, \bullet\}$$

If's e whiles.

$$\frac{\Gamma \vdash b :: \mathcal{Q}_1\ \mathsf{Bool} \quad \Gamma \models_\tau c_1 :: (\mathcal{Q}_2\ \tau, \Gamma') \quad \Gamma \models_\tau c_2 :: (\bullet, \Gamma'')}{\Gamma \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \Gamma)}$$

$$\frac{\Gamma \vdash b :: \mathcal{Q}_1\ \mathsf{Bool} \quad \Gamma \models_\tau c_1 :: (\bullet, \Gamma') \quad \Gamma \models_\tau c_2 :: (\mathcal{Q}_2\ \tau, \Gamma'')}{\Gamma \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \Gamma)}$$

$$\frac{\Gamma \vdash b :: \mathcal{Q}\ \mathsf{Bool} \quad \Gamma \models_\tau c_1 :: (\bullet, \Gamma') \quad \Gamma \models_\tau c_2 :: (\bullet, \Gamma'')}{\Gamma \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet, \Gamma)}$$

$$\frac{\Gamma \vdash b :: \mathcal{Q}_1\ \mathsf{Bool} \quad \Gamma \models_\tau c_1 :: (\mathcal{Q}_2\ \tau, \Gamma') \quad \Gamma \models_\tau c_2 :: (\mathcal{Q}_2\ \tau, \Gamma'')}{\Gamma \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\mathcal{Q}_2\ \tau, \Gamma)}$$

$$\frac{\Gamma \vdash b :: \mathcal{Q}_1\ \mathsf{Bool} \quad \Gamma \models_\tau c :: (\rho, \Gamma')}{\Gamma \models_\tau \mathsf{if}\ b\ \{c\}\ :: (\bullet, \Gamma)} \quad \rho \in \{\tau, \bullet\}$$

$$\frac{\Gamma \vdash b :: \mathcal{Q}_1\ \mathsf{Bool} \quad \Gamma \models_\tau c :: (\rho, \Gamma')}{\Gamma \models_\tau \mathsf{while}\ b\ \{c\}\ :: (\bullet, \Gamma)} \quad \rho \in \{\tau, \bullet\}$$

A função *fQual*, apresentada ao longo das diversas regras, tem por objectivo calcular o conjunto de qualificadores que é propagado em cada uma dessas regra. Desta forma, para cada regra e cada qualificador define-se uma função $fQual^{q_{id}}_{r_{id}}$ que determina se o qualificador q_{id} é ou não inferido na regra r_{id}, ou seja, cada função $fQual^{q_{id}}_{r_{id}}$ é responsável por verificar se o qualificador sobre o qual esta definida pertence ao conjunto de qualificadores que é obtido no final da regra. Note-se ainda que cada uma destas funções é apenas responsável pela eventual ocorrência do qualificador q_{id} nesse conjunto final. Desta forma todos os restantes qualificadores presentes nos conjuntos que a função recebe como argumento não são considerados.

Esta função funciona neste contexto como uma "meta-função" uma vez que o seu resultado depende unicamente do resultado de cada uma das funções $fQual^{q_{id}}_{r_{id}}$. Além disso, pode receber um número diferente de argumentos e, consoante esse número invoca apenas as funções $fQual^{q_{id}}_{r_{id}}$ cuja aridade é igual à sua. No caso concreto das regras que acabamos de apresentar a função *fQual* pode receber como argumento um ou dois conjuntos de qualificadores dependendo da regra na qual está a ser utilizada. Desta forma, a função *fQual* define-se, para qualquer qualificador q_{id} e regra r_{id}, da seguinte forma:

$$fQual :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual\ \mathcal{Q} \;=\; \bigcup_{\{q\}\ \in\ \mathcal{Q}} \left(\{q\} \;=\; fQual^{q_{id}}_{r_{id}}\ \mathcal{Q}\right)$$

$$fQual :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual\ \mathcal{Q}_1\ \mathcal{Q}_2 \;=\; \bigcup_{\{q\}\ \in\ \mathcal{Q}'} \left(\{q\} \;=\; fQual^{q_{id}}_{r_{id}}\ \mathcal{Q}_1\ \mathcal{Q}_2\right)$$

onde, $\mathcal{Q}' \;=\; \mathcal{Q}_1 \;\cup\; \mathcal{Q}_2$

Cada função $fQual^{q_{id}}_{r_{id}}$ devolve um conjunto singular (com o qualificador q_{id}) ou vazio e portanto a função *fQual* calcula o conjunto formado pela união dos conjuntos devolvidos por cada uma das funções auxiliares que invoca. Finalmente, e como dissemos anteriormente, a utilização da função *fQual* tem como objectivo evitar a necessidade de definir a mesma regra para cada qualificador. Assim, ao longo deste documento vamos apresentar a definição das várias funções $fQual^{q_{id}}_{r_{id}}$ à medida que são apresentados os diferentes qualificadores.

3.6.2 Inferência dos Qualificadores

Aquando de cada utilização de qualquer qualificador é necessário verificar em que circunstâncias as variáveis por si qualificadas podem ou não ser correctamente utilizadas. No caso dos qualificadores definidos pelo utilizador é necessário ter em conta duas situações distintas, onde intervêm respectivamente, qualificadores definidos com a cláusula `case` e qualificadores definidos com a cláusula `assume`.

Ao longo desta secção vamos apresentar as diversas situações concretas em que os qualificadores definidos pelos utilizadores são inferidos. Na prática, estes não são inferidos (no sentido tradicional do termo) mas sim propagados das declarações para as expressões. É a esta propagação que chamamos de inferência dos qualificadores.

3.6.2.1 Clausula `assume`

Este é o caso mais simples, que tal como vimos anteriormente, é utilizado para definir qualificadores cuja responsabilidade da sua utilização é exclusiva do programador, não existe qualquer verificação por parte do sistemas de tipos relativa à sua consistência. Exceptuando os *asignments* e os *casts*, qualquer variável cujo tipo esteja qualificado com um qualificador `assume` pode ser utilizada nas mesmas circunstâncias em que o seria caso o tipo não fosse qualificado, ou seja, nestas situações, o qualificador não expressa qualquer tipo de restrição na utilização das variáveis.

Nestes casos, quando estas variáveis ocorrem em qualquer expressão o qualificador é ignorado pelo sistema de tipos, isto acontece porque as funções $fQual^{q_{id}}_{r_{id}}$ responsáveis por verificar se estes qualificadores são propagados pelas regras de tipagem do CAO devolvem sempre um conjunto vazio, isto é, estes qualificadores nunca são inferidos pelas regras.

$$fQual^{q_{id}}_{r_{id}} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{q_{id}}_{r_{id}}\ \mathcal{Q}_1\ =\ \{\}$$

onde, q_{id} é um qualificador definido por uma cláusula `assume` e r_{id} é uma regra de tipagem.

$$fQual^{q_{id}}_{r_{id}} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual^{q_{id}}_{r_{id}}\ \mathcal{Q}_1\ \mathcal{Q}_2 =\ \{\}$$

onde, q_{id} é um qualificador definido por uma cláusula `assume` e r_{id} é uma regra de tipagem.

Como referimos anteriormente, as duas únicas situações onde os qualificadores `assume` expressam restrições na utilização da variáveis por si qualificadas estão relacionadas com os *assignments* e com os *casts*. No primeiro caso, quando uma variável é declarada em Γ com um qualificador `assume` então, quando usada individualmente num *assignment* como *LValue* ou *RValue* o qualificador é preservado. No segundo caso, como referimos na Secção 3.5.3.2, é sempre possível realizar um *cast* para um tipo qualificado mesmo que a definição do qualificador não garanta que a expressão em causa cumpra de facto com a propriedade associada a esse qualificador. Nestes casos a responsabilidade é assumida pelo programador e portanto

o qualificador é sempre obtido na conclusão da regra. **Variáveis**

$$fQual_{var}^{q_{id}} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual_{var}^{q_{id}}\ \mathcal{Q}_1\ =\ \{q_{id}\}\ \cap\ \mathcal{Q}_1$$

Casts

$$fQual_{()}^{q_{id}} :: \mathcal{Q}_1 \rightarrow\ \mathcal{Q}_2\ \rightarrow\ \mathcal{Q}$$

$$fQual_{()}^{q_{id}}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ \{q_{id}\}\ \cap\ \mathcal{Q}_2$$

Vejamos os seguintes exemplos de utilização do qualificador `prime` definido anteriormente:

```
def x, y, z : prime int;
def w: int;
x := y + z;                 1
x := y;                     2
x := (prime int) y + w;     3
```

No exemplo 1 o *assignment* não é válido porque a variavel x tem o tipo `prime int` e o tipo da expressão $y + z$ é `int` isto porque, embora as suas duas sub-expressões sejam `prime int`, pela definição da função $fQual_{+}^{\texttt{prime}}$ sabemos que o qualificador não é propagado pela regra de tipagem desta operação. No segundo exemplo, o `assignment` envolve duas variáveis `prime int` e como os qualificadores `assume` são preservados pela regra de tipagem das variáveis o *assignment* é efectuado com sucesso. Também o `assignment` do exemplo 3 é efectuado com sucesso uma vez que embora a expressão usado como *RValue* tenha tipo `int` e a variável do *LValue* seja `prime int`, o *cast* utilizado, que funciona como um assumir de responsabilidade por parte do programador, permite que esta operação seja concluída.

3.6.2.2 Clausula `case`

Este é o caso mais complexo, pois de entre os qualificadores definidos pelo utilizador os qualificadores `case` são aqueles com os quais é possível definir diversas

restrições sobre os tipos da nossa linguagem. Estas restrições podem ser estabelecidas em duas situações distintas, através da qualificação de um tipo já qualificado, caso onde se está a estabelecer uma ordem na qualificação, como vimos na Secção 3.5.3.1, ou através da definição de regras de tipagem na definição do qualificador. Nestas regras as restrições são estabelecidas em duas partes, no padrão da regra e na expressão booleana da respectiva condição lateral. Desta forma, em cada utilização de uma variável qualificada com um qualificadore `case` num *assignment* ou *return* de função é necessário verificar em primeiro lugar o padrão e caso este seja consistente avaliar a condição lateral, e só nos casos em que esta avalia em `true` a utilização do qualificador é correcta.

A primeira parte desta verificação é baseada num mecanismo de "match" entre a expressão utilizada no *assignment* e os padrões definidos nas regras de tipagem. Desta forma, quando num *assignment* é atribuído a uma variável cujo tipo é qualificado por um qualificador `case`, se na definição desse qualificador não existir um padrão que combine com a expressão utilizada como *RValue* no *assignment* este é de imediato rejeitado. Consideremos o qualificador `positive` definido na Figura 3.5 e o seguinte excerto de código:

```
def x: positive int;
def y, z : int;
x := y - z;   1
x := y + z;   2
```

No exemplo 1 o *assignment* é de imediato rejeitado uma vez que a expressão $y - z$ não faz "match" com nenhum dos padrões estabelecidos na definição do qualificador. No segundo exemplo como na definição do qualificador existe um padrão que faz "match" com a expressão $y + z$ esta fase da verificação é concluída e passa-se então à avaliação da condição lateral da regra de tipagem onde este padrão ocorre.

Match

O algoritmo que verifica se existe algum padrão na definição do qualificador que combina com a expressão utilizada tem em conta dois aspectos, o operador utilizado em ambas as situações e o tipo dos operandos. Dada uma expressão utilizada num *assignment*, são percorridas de forma sequencial as várias regras de tipagem que definem o qualificador, quando é encontrado um padrão com uma operação

coincidente com a operação definida na expressão é então verificada a condição lateral dessa regra de tipagem. A partir do momento em que é encontrado este padrão a fase de "macth" do algoritmo termina.

Avaliação da condição lateral

O algoritmo de avaliação das condições laterais de cada regra de tipagem utiliza algumas propriedades da lógica proposicional [19] para decidir se a condição é ou não válida. Como vimos anteriormente, em cada condição lateral podemos ter apenas expressões booleanas envolvendo meta-variáveis, literais e os operadores $==$, $\leq$, $\geq$, $!=$, invocação de qualificadores, e conjunções, disjunções e negação destas expressões. Recordemos a sintaxe das condições laterais, definida na Figura 3.3.

$$
\begin{array}{rcl}
sc & ::= & e_1 \oplus e_2 \mid sc \;\&\&\; sc \mid sc \parallel sc \mid !sc \mid q_{id}(x) \mid b_{lit} \\
b_{lit} & ::= & true \mid false \\
\text{com,} & & \\
\oplus & \in & \{\geq,\ \leq,\ >,\ >,\ ==\}
\end{array}
$$

Assim, para qualquer condição lateral sc as meta-variáveis que nela ocorrem são instanciadas pelos valores ou variáveis associados à expressão que está a ser utilizada, originando desta forma uma condição sc' que é avaliada pela pela função $[\![sc']\!]$ que se define da seguinte forma:

$$
\begin{array}{rcl}
[\![.]\!] & : & sc \rightarrow Bool_{\perp} \\
[\![e_1 \oplus e_2]\!] & = & \texttt{val}(e_1) \oplus \texttt{val}(e_2) \\
[\![sc_1 \;\&\&\; sc_2]\!] & = & [\![sc_1]\!] \;\&\&\; [\![sc_2]\!] \\
[\![sc_1 \parallel sc_2]\!] & = & [\![sc_1]\!] \parallel [\![sc_2]\!] \\
[\![!sc]\!] & = & \neg [\![sc]\!] \\
[\![q_{id}(x)]\!] & = & \texttt{isDefined}(q\_id,\ \texttt{getType}(x))) \\
[\![true]\!] & = & true \\
[\![false]\!] & = & false
\end{array}
$$

Onde `val` é a função que dada uma expressão devolve o seu respectivo valor associado. Quando estas expressões são formadas apenas por literais (ou meta-variáveis que representam literais) esta função usa a função ϕ_l definida na Secção 3.6 para avaliar as expressões. Quando nas expressões ocorrem meta-variáveis que representam variáveis indeterminadas é gerado um erro a informar que a condição lateral não pôde ser avaliada. Isto porque, como referimos anteriormente, Finalmente, `isDefined` é a função que dado o identificador de um qualificador e o identificador de uma variável (ou um literal) determina se esse qualificador foi

previamente definido sobre o tipo dessa variável (aqui determinado pela função `getType`).

Regras de tipagem. Relativamente aos qualificadores definidos com a cláusula `case` é necessário ter em conta que, ao contrário do que acontece com os restantes qualificadores, as regras usadas durante a tipagem de expressões envolvendo estes qualificadores dependem do conjunto de cláusulas `case` estabececidas na definição de cada qualificador. Por esta razão o resultado das função *fQual* depende também deste conjunto de clausulas estabelecidas aquando da definição do qualificador.

Por exemplo, recordemos a definição do qualificador `positive` definido na Figura 3.5. Este qualificador é definido através de três cláusulas `case`. Usando a mesma representação que usamos para tipar as expressões com qualificadores definidos com a cláusula assume podemos reescrever as regras de tipagem estabelecidas na definição do qualificador `positive` da seguinte forma:

A primeira cláusula é tipada pela seguinte regra:

$$\frac{\Gamma(i) = \mathcal{Q}\ \mathsf{Int} \quad \phi_l(i) > 0}{\Gamma \vdash i :: (fQual\ \mathcal{Q})\ \mathsf{Int}}\, i\ \in\ \mathbb{Z}$$

A segunda e terceira cláusulas são tipadas pela regra:

$$\frac{\Gamma_G \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \mathsf{Int}} \quad \oplus \in \{+, *\}$$

Onde *fQual* é a função definida na Secção 3.6.1 e as funções auxiliares $fQual_{lit}^{\texttt{positive}}$ e $fQual_{\oplus}^{\texttt{positive}}$, responsáveis por determinar se o qualificador `positive` é inferido em cada uma destas regras definem-se, respectivamente, da seguinte forma:

$$fQual_{lit}^{positive} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual_{lit}^{positive}\ \mathcal{Q}_1\ = \{\texttt{positive}\}$$

$$fQual_{\oplus}^{positive} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{\oplus}^{positive}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{positive}\}\ \cap\ \mathcal{Q}_1\ \cap\ \mathcal{Q}_2)$$

onde $\oplus \in \{+, *\}$

Além disso, todos os qualificadores `case` são preservados pela regra de tipagem das variáveis (que ocorrem como *LValue*). Desta forma, a funções $fQual^{q_{id}}_{var}$ onde q_{id} é um qualificador definido por uma cláusula `case` define-se na seguinte forma:

Variáveis

$$fQual^{q_{id}}_{var} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{q_{id}}_{var}\ \mathcal{Q}_1\ =\ \{q_{id}\}\ \cap\ \mathcal{Q}_1$$

Assim, o seguinte *assignment*,

```
def x : positive int;
x := 4 + 3;
```

é válido porque ambos os lados têm um único qualificador (`positive`), e portanto a condição $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$ da regra do *assignment* é respeitado. O *LValue* é `positive int` porque a variável `x` foi declarada com este tipo e a regra de tipagem das variáveis *LValue* preserva este qualificador. O *RValue* é também `positive int` porque é uma soma entre dois literais (que são `positive int`) e pela regra $\oplus$ a soma de dois `positive int` é ainda `positive int`. Finalmente, os literais `3` e `4` são `positive int` porque, são literais de inteiros e $\phi_l(4) = 4 > 0$ e $\phi_l(3) = 3 > 0$.

Finalmente, e tal como acontece com os qualificadores `assume`, é possível realizar *casts* para tipos qualificados. Funcionando estes *casts* também aqui como um assumir de responsabilidade por parte do programador. Desta forma, a função $fQual^{q_{id}}_{()}$ responsável por determinar se o qualificador q_id é inferido na regra de tipagem dos *casts* define-se da seguinte forma:

Casts

$$fQual^{q_{id}}_{()} :: \mathcal{Q}_1 \rightarrow\ \mathcal{Q}_2\ \rightarrow\ \mathcal{Q}$$

$$fQual^{q_{id}}_{()}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ \{q_{id}\}\ \cap\ \mathcal{Q}_2$$

```
1         def gcd (a, b: positive int): poisitve int {
2             def c: positive int;
3             while(a % b != 0){
4                 c := (positive int) a % b;
5                 a := b;
6                 b := c;
7           }
8             return b;
9         }
10
11        def lcm(a,b : positive int): positive int{
12           def d: positive int := gcd(a,b);
13           def p: positive int := a*b;
14           return (positive int) (p / d);
15        }
```

FIGURA 3.7: Funções `gcd` e `lcm` definidas em CAO.

3.7 Exemplos de Utilização

Nesta secção vamos apresentar alguns exemplos de programas escritos em CAO que utilizam alguns dos qualificadores referidos anteriormente. O primeiro exemplo na Figura 3.7 contém a codificação da função que calcula o máximo divisor comum entre dois inteiros. O algoritmo implementado é um dos mais utilizados para calcular esta função. Esta recebe como argumento duas variáveis com o tipo `positive int` e retorna um terceiro `positive int`. Esta implementação tem a particularidade de exigir (linha 4) que o resto da divisão inteira entre os dois parâmetros da função seja `positive int`. No entanto, como na definição do qualificador `positive` não existe nenhuma regra de tipagem que garanta que o resultado desta operação seja ainda um `positive int` é necessário definir explicitamente um *cast*. Se o *cast* não fosse utilizado o tipo do *RValue* do *assignment* seria `int` porque, tanto `a` como `b` são coerciveis para o seu supertipo (`int`) e portanto o tipo resultante da operação % seria ainda `int`. Assim, teríamos um *assignment* onde o tipo do *LValue* é diferente do tipo do *RValue* e portanto seria gerado um erro de tipagem.

Ainda na Figura 3.7, o segundo exemplo contém o código da codificação de uma função que calcula o mínimo múltiplo comum entres duas variáveis `positive int` e cujo resultado é ainda `positive int`. Neste exemplo o *assignment* da linha 12 é possível porque a função `gcd` devolve um `positive int` e portanto temos

um *assignment* entre duas expressões com o mesmo tipo, o mesmo acontece na linha 13 onde a instrução envolve uma variável `positive int` e uma operação binária em que os dois operandos são também inteiros positivos e pela definição deste qualificador, o produto entre dois `positive int` é ainda `positive int`. Finalmente, na linha 14, tal como no *assignment* da linha 4, é necessário recorrer a um *cast* uma vez que na definição do qualificador não existe qualquer regra que faça "match" com a expressão que esta a ser retornada.

3.8 Trabalho Relacionado

Nesta secção vamos apresentar e fazer uma breve discussão sobre trabalho baseados nos conceitos de qualificação de tipo existentes na literatura.

Como referimos anteriormente, um dos grandes objectivos de qualquer linguagem de programação é permitir que os programadores consigam expressar as propriedades desejáveis do comportamento dos programas. No entanto, os sistemas de tipos tradicionais não conseguem capturar muitas dessas propriedades. Por essa razão, há um grande interesse em refinar os tipos convencionais das linguagens de programação de forma a que seja possível expressar estas propriedades.

Nesse sentido, Freeman e Pfenning [20][21] usam refinamento de tipos para expressar restrições sobre estruturas de dados recursivas definidas algebricamente em programas ML. Esta teoria de refinamento de tipos foi posteriormente enriquecida por Xi e Pfenning [22] que desenvolveram um sistema designado por "Dependent ML" bastante mais complexo e que explora a capacidade de os tipos dependentes raciocinarem sobre invariantes em estruturas de dados complexas.

Uma abordagem alternativa sem a utilização explícita de refinamento de tipo foi apresentada por EvansčiteQEVA96 que introduziu um conjunto de qualificadores baseados em anotações à linguagem C, um desses qualificadores é o `nonnull` utilizado para garantir que um apontador nunca tenha o valor `null`. Apesar das suas vantagens, estas ferramentas baseadas em anotações têm bastantes limitações que as tornam pouco usáveis. Estas utilizam um conjunto fixo de regras de tipagem e embora possam, dentro de um certo limite, simular as regras dos refinamentos de tipos estas não são expressivas o suficiente para lidar com muitas situações comuns. Por exemplo, é impossível simular uma regra de tipagem de uma expressão se esta depender recursivamente do tipo de sub-expressões.

Para resolver estes problemas, Jeffrey Foster desenvolveu uma ferramenta chamada *CQual*[1] [11] que permite que se adicionem qualificadores de tipos definidos pelo utilizador a linguagens como *C*, *C++*, *Java* ou *ML*. Esta ferramenta pode ser usada para adicionar qualificadores *flow-insensitive* [23] e *flow-sensitive* [24] os primeiros são associados às variáveis de um programa, os segundos são associados a um determinado local de memória que pode ser alterado a qualquer momento.

Para resolver problemas similares Chin, Markstrum and Millstein [25][26] desenvolveram outra ferramenta, chamada *Clarity* que fornece uma linguagem onde os utilizadores podem escrever as regras de tipagem dos novos qualificadores, o que permite a definição de propriedades mais expressivas do que as que se definem recorrendo apenas às anotações. Estas novas regras de tipagem são automaticamente incorporadas na verificação estática de tipos da linguagem. Além disso, este sistema utiliza ferramentas de prova para verificar a consistência da especificação dos qualificadores. Ao contrário do *CQual*, esta ferramenta não inclui inferência de tipos.

[1] Disponivel de forma gratuita em http://cqual.sourceforge.net.

Capítulo 4

Qualificador Const

4.1 Introdução

No Capítulo 3 mostramos como os qualificadores são extremamente úteis para estabelecer diversas propriedades sobre os tipos de linguagens de programação. Neste capítulo vamos usar qualificadores para definir a noção de variável constante em CAO.

Em diversas linguagens de programação, devido à necessidade de usar parâmetros constantes em diversas situações, existe a possibilidade de declarar variáveis como sendo constantes. Em linguagem como o C ou C++ esta propriedade é obtida através do qualificador `const`. Neste caso, o qualificador é usado para garantir que variáveis (ou posições de memória) por si qualificadas não são alteradas após a sua inicialização. Por exemplo, o seguinte protótipo de função em C,

$$f(\texttt{const}\ int\ *x)$$

garante que a função f não vai escrever na posição de memória apontada por x.

Também no *Java* é possível estabelecer este tipo de restrições sobre as variáveis. Para tal a linguagem dispõem do qualificador `final`. Consideremos a classe `Point` em *Java* para representar pontos em duas dimensões. Esta classe tem as variáveis de instância `x` e `y`. Se declararmos uma instância da classe `Point` com o qualificador `final`,

$$public\ \texttt{final}\ Point\ pos;$$

como `pos` é uma variável `final` não lhe pode ser atribuído valor, no entanto, individualmente é possível atribuir valor às variáveis `pos.x` e `pos.y`.

Tal como em C ou em *Java* também no CAO definir variáveis constantes é algo de extrema utilidade.

Como referimos no Capítulo 2 aquando da declaração de vectores em CAO o parâmetro que define o seu tamanho tem que obrigatoriamente ser um literal inteiro. Algo semelhante acontece nos *ranges* onde os valores que definem os limites do intervalo seleccionado têm também que, obrigatoriamente, ser literais. É aqui que a noção de constante em CAO se torna uma mais-valia. Com constantes torna-se possivel declarar vectores cujo tamanho pode depender, além de literais, de expressões constantes. Nos *ranges* acontece algo similar, onde os limites que definem o intervalo passam também a poder ser, além de literais, expressões constantes.

Outro caso onde a noção de expressão constante é de extrema utilidade está relacionado com a implementação de técnicas criptográficas. Em muitas primitivas criptográficas é necessário utilizar parâmetros que têm que ser instanciados por um valor em concreto e que são utilizados em várias situações distintas ao longo da primitiva criptográfica. Ao declararmos estes parâmetros como constantes não temos a necessidade de alterar todas as suas ocorrências quando a primitiva for utilizada com um parâmetro diferente. Além disso, sempre que os parâmetros são declarados como constantes não podem ser alterados ao longo da primitiva, algo que as variáveis tradicionais (não constantes) não conseguem garantir.

4.2 Reconhecimento de Expressões Constantes

No Capítulo 3 mostramos o mecanismo que permite adicionar qualificadores de tipos definidos pelo utilizador à linguagem CAO. Utilizando este mecanismo podemos em particular definir qualificadores que permitam raciocinar sobre a noção de constante em CAO e desta forma restringir o comportamento dos programas. Para tal é necessário identificar inequivocamente o conjunto de expressões que podemos considerar constantes.

A noção de expressão constante é bastante dúbia e pode ter diversas interpretações, podemos ter expressões constantes simples, formadas apenas por literais e operações

sobre literais. Expressões onde além de literais ocorrem variáveis constantes ou, indo um pouco mais longe, expressões com literais, variáveis constantes e invocação de funções. No entanto, esta última alternativa acarreta um problema adicional, que consiste em identificar as funções que podem ser usadas em expressões constantes.

Para que a invocação de funções não comprometa a característica fundamental das expressões constantes, relacionada com o facto de estas poderem ser avaliadas em tempo de compilação, estas funções devem respeitar as seguintes condições:

- Término: é necessário garantir que as funções usadas nas expressões constantes terminam.
- Cada função, invocada várias vezes com os mesmos argumentos deverá devolver sempre os mesmos resultados, portanto, não podem depender de valores (pseudo)-aleatórios nem de variáveis globais (excepto variáveis globais constantes).
- Todas as funções que estas possam usar têm também que respeitar estas características.

A estas funções chamamos de funções puras. E apenas estas devem ocorrer em expressões constantes.

Para exemplificar a definição e utilização destes qualificadores que estabelecem a noção de constante em CAO vamos utilizar apenas o tipo `int` embora estes qualificadores possam naturalmente ser definidos sobre outros tipos da linguagem.

O qualificador `const` sobre o tipo `int` pode ser definido através de cláusulas `case` que estabelecem o conjunto de situações em que variáveis constantes inteiras podem ser instanciadas. Por exemplo, podemos definir um qualificador `const` que permite que às variáveis constantes sejam atribuídos apenas literais ou expressões aritméticas envolvendo literais.

```
qualifier const(x: int){
 case x of
  def A : int;
  lit A, where true;
  def A, B : int;
```

```
        lit A + lit B, where const<A> && const<B>;
    }
```

Este qualificador, definido com duas cláusulas `case`, permite que as expressões constantes no CAO sejam formadas por literais ou soma de literais. No entanto a esta definição podem ser adicionadas novas cláusulas de forma a permitir outras expressões aritmeticas. Além disso, tal como mostramos no Capítulo 3, cada clausula `case` origina uma regra de tipagem que por sua vez dá origem a uma função auxiliar $fQual_{r_{id}}^{q_{id}}$.

A sintaxe da declaração de variáveis com este qualificador não sofre qualquer alteração relativamente à sintaxe referente aos restantes qualificadora definidos pelo utilizador. Com este qualificador `const`, é possível atribuir valores literais a variáveis declaradas como `const int`.

```
def x: const int := 4;
```

Sem a utilização de literais a única forma de inicializar uma variável constante é através de um *assignment* envolvendo uma segunda variável constante, como podemos ver no seguinte excerto de código CAO.

```
def x: const int := 4;
def y: const int := x;
```

4.2.1 Sistema de tipos

Além do qualificador `const`, para se estabelecer a noção de constante pretendida é ainda necessário garantir que as variáveis declaradas como este qualificador não são alteradas ao longo dos programas, ou seja, temos que garantir que as variáveis são instanciadas uma única vez. Para tal é necessário alterar o sistema de tipos da linguagem, de forma a diferenciar as variáveis `const` das restantes variáveis.

No CAO optamos por uma implementação onde as variáveis `const` têm que obrigatoriamente ser inicializadas no momento da sua declaração. Para tal, são necessárias duas alterações ao sistema de tipo.

Como mostramos no Capítulo 2 a declaração de variáveis em CAO pode ser realizada de duas formas distintas, com declaração e inicialização numa única instrução ou apenas com declaração. Assim, a primeira alteração ao sistema de tipos da linguagem visa precisamente impedir que variáveis `const` sejam declaradas numa instrução sem inicialização. Note-se que, alternativamente, poderíamos optar por uma versão onde as variáveis `const` poderiam ser declaradas sem inicialização ficando estas com um valor por defeito que no caso dos inteiros poderia ser o valor zero. A segunda alteração ao sistema de tipos tem como objectivo impedir *assignments* cujos *LValues* sejam varáveis `const`.

Com estas alterações, e como o qualificador `const` tem especificidades que o diferenciam dos restantes, é necessário que o sistema de tipos o consiga distinguir de forma garantir que as restrições que estas alterações trazem são respeitadas. Assim, ao contrario dos restantes qualificadores o `const` está predefinido na linguagem e portanto não é possível declarar novos qualificadores sobre o tipo `int` com o identificador `const`.

Formalmente as duas alterações ao sistema de tipos do CAO originam as seguintes alterações nas regras de tipagem apresentadas na Secção 3.6.1.

Declarações.

$$\frac{}{\Gamma \models_{\rho} \mathsf{def}\ x : \mathcal{Q}\ \tau :: (\bullet, \Gamma[x :: \mathcal{Q}\ \tau])} \quad x \notin dom(\Gamma),\ \texttt{const} \notin \mathcal{Q}$$

Na declaração sem inicialização, com a restrição $\texttt{const} \notin \mathcal{Q}$ passa a não ser possível declarar variáveis com o qualificador `const`.

Algo semelhante acontece na declaração de funções onde tanto os tipos dos argumentos como os tipos de retorno não podem ser `const`.

$$\frac{\Gamma, \epsilon[x_1 :: \mathcal{Q}_1\ \tau_1, \ldots, x_n :: \mathcal{Q}_n\ \tau_n] \models_{\tau} c :: (\mathcal{Q}\ \tau, \Gamma'_G)}{\Gamma, \epsilon \models \mathsf{def}\ fp(x_1 : \mathcal{Q}_1\ \tau_1, \ldots, x_n : \mathcal{Q}_n\ \tau_n) : \mathcal{Q}'\ \tau'\ \{c\} :: (\bullet, \Gamma[fp :: ((\mathcal{Q}_1\ \tau_1, \ldots, \mathcal{Q}_n\ \tau_n) \rightarrow \mathcal{Q}'\ \tau')])}$$

onde, $\tau' \neq \mathsf{void}$, $fp \notin dom(\Gamma)$ e $\texttt{const} \notin (\mathcal{Q}'\ \cup\ \mathcal{Q}_1 \cup , \ldots, \cup\ \mathcal{Q}_n)$.

Nesta nova regra a restrição $\texttt{const} \notin (\mathcal{Q}' \cup \mathcal{Q}_1 \cup , \ldots, \cup\ \mathcal{Q}_n)$ impede que qualquer argumento ou valor de retorno da função seja `const`. Para as funções que retornam

`void` define-se uma restrição análoga à anterior cuja exclusão do qualificador `const` incide apenas sobre o conjuntos de qualificadores dos argumentos.

As regras relativas à declaração de novos tipos e à declaração com inicialização de variáveis não sofrem qualquer alteração.

LValues.

$$\frac{\Gamma(x) = \mathcal{Q}\ \tau}{\Gamma \vdash x :: (fQual\ \mathcal{Q})\ \tau} \quad \tau \in dom(\Gamma),\ \texttt{const} \notin \mathcal{Q}$$

Nesta regra a restrição `const` $\notin \mathcal{Q}$ impede que variáveis `const` ocorram como *LValue*. As restantes regras dos *LValues* não são alteradas porque estamos a considerar apenas a o qualificador `const` sobre os inteiros.

4.3 Avaliação de Expressões Constantes nos Tipos

A versão do qualificador `const` apresentada na secção anterior, apesar de nos permitir declarar variáveis constantes, não oferece uma solução para o problema que pretendemos solucionar. Permitir que na declaração de vectores possam ser usadas expressões constantes e não apenas literais de inteiros. Assim, é necessário enriquecer a noção de constante do CAO. Esta versão enriquecida deverá ser capaz de permitir expressões constantes que possam ser avaliadas em tempo de compilação para assim serem usadas pelo sistema de tipos da linguagem.

Para tal, é fundamental que em cada momento o sistema de tipos consiga determinar o valor de cada uma das variáveis declaradas com o qualificar `const`. Para resolver este problema optamos por uma solução onde, em vez de um qualificador que funciona apenas com uma "flag" que nos indica que uma qualquer variável é constante, cada qualificador tem a si associado um determinado valor. Na prática em vez de um qualificador `const` temos vários qualificadores `const-n` onde `n` é o valor inteiro que cada um tem a si associado. Para os restantes tipos do CAO (diferentes de `int`) acontece algo similar. No entanto, vamos apenas apresentar a caso particular do qualificador `const` que actua sobre inteiros, pois, este é o tipo sobre o qual os qualificadores `const` têm maior utilidade no contexto deste trabalho, nomeadamente, na declaração e *ranges* de vectores.

Como referimos na Secção 4.2 existem diversas alternativas para aquilo a que chamamos expressões constante. De entre essas alternativas, apresentadas anteriormente, a invocação de funções é a única que actualmente não é considerada pelo sistema de qualificação do CAO. Isto porque para que tal fosse possível seria necessário estabelecer um mecanismo de qualificação (ou algo similar) sobre as funções de forma a distinguir funções puras das restantes funções. Assim, no CAO expressões inteiras constantes são formadas unicamente por literais, variáveis `const` e operações aritméticas envolvendo estes literais e variáveis.

4.3.1 Sintaxe

A sintaxe do CAO que apresentamos na Figura 2.1 obriga a que os parâmetros utilizados tanto na declaração como nos *ranges* de vectores sejam literais inteiros. Desta forma, e como o principal objectivo da introdução do qualificador `const` é permitir que estes sejam definidos através de expressões constantes, é necessário alterar a sintaxe da linguagem de forma a aceitar expressões nestes parâmetros.

Relativamente às restantes construções do CAO a sua sintaxe permanece inalterada. Também a introdução dos qualificadores `const-n` não origina qualquer alteração na sintaxe da linguagem. Isto porque sintacticamente existe apenas o qualificador `const` e portanto é apenas com este que os programadores lidam. Os qualificadores `const-n` correspondem às representações de cada qualificador `const` e são usados unicamente pelo sistema de tipos, sendo totalmente invisíveis para os programadores.

4.3.2 Sistema de Tipos

Em qualquer linguagem que disponha de qualificadores de tipos um dos factores mais importante está relacionada com a inferência dos qualificadores. É necessário identificar claramente todas as situações concretas em que os qualificadores são inferidos pelas construções da linguagem. Nesta secção vamos apresentar as várias situações concretas em que o qualificador `const` é inferido pelas regras de tipagem do CAO.

No conjunto de regras de tipagem do CAO que apresentamos na Secção 3.6.1 a referência aos qualificadores é feita de forma global, isto é, não são discriminados individualmente cada um dos qualificadores. Em cada regra intervêm vários conjuntos genéricos de qualificadores que são manipulados por uma função $fQual$ responsável por calcular o conjunto de qualificadores inferidos nessa regra. Esta função $fQual$ utiliza uma serie de funções auxiliares, $fQual^{q_{id}}_{r_{id}}$, responsáveis por determinar se o qualificador q_{id} é inferido na regra r_{id} assim, cada função auxiliar devolve um conjunto vazio ou um conjunto singular com o qualificador sobre o qual está definida.

4.3.2.1 Inferência do Qualificador `const`

Nesta secção vamos apresentar a definição das várias funções auxiliares $fQual^{const}_{r_{id}}$. Esta apresentação segue a mesma ordem pela qual as regras foram introduzidas na secção 3.6.1. Para cada regra r_{id} define-se uma função $fQual^{const}_{r_{id}}$. Para simplificar a sua apresentação, utilizamos uma definição para estas funções baseadas na teoria de conjuntos [27].

Como internamente utilizamos vários qualificadores `const-n` para representar o qualificador `const`, o sistema de tipos da linguagem tem que sofrer algumas alterações de forma a lidar com esta novo facto. Assim, utilizamos a palavra `const` quando nos referimos ao qualificador `const` sintáctico e as palavras `const-n` ou `const-m` para as várias representações que este pode tomar e que são reconhecidas apenas internamente pelo sistema de tipos da linguagem. Ao longo da definição das funções $fQual^{const}_{r_{id}}$ utilizamos ainda a notação $\{\texttt{const-n}\}$ para representar o conjunto singular com o qualificador `const-n`.

Variáveis, chamadas de funções e projecções de estruturas.

$$fQual^{const}_{var} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{const}_{var}\ \mathcal{Q}_1 \ = \ (\{\texttt{const-n}\} \ \cap \ \mathcal{Q}_1)$$

para algum $\mathtt{n} \in \mathbb{Z}$.

Se uma variável em Γ tem um tipo qualificado com `const-n` então sempre que esta for usada numa expressão o seu tipo permanecerá com este qualificador.

$$fQual^{const}_{fCall} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{const}_{fCall}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ \{\}$$

A chamada de funções não pode ser usada em expressões constantes, por esta razão a função $fQual^{const}_{fCall}$ devolve sempre um conjunto vazio.

$$fQual^{const}_{.fi} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual^{const}_{.fi}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ \{\}$$

Tal como o retorno de funções, os acessos a campos de *structs* não podem ser utilizados em expressões constantes, assim, a função $fQual^{const}_{.fi}$ devolve sempre um conjunto vazio.

Operadores aritméticos.

$$fQual^{const}_{\oplus} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual^{const}_{\oplus}\ \mathcal{Q}_1\ \mathcal{Q}_2\ = \begin{cases} \texttt{const-n} \in \mathcal{Q}_1\ \&\&\ \texttt{const-m} \in \mathcal{Q}_2 & \Rightarrow \{\texttt{const-n'}\} \\ \texttt{const-n} \notin \mathcal{Q}_1\ \|\ \texttt{const-m} \notin \mathcal{Q}_2 & \Rightarrow \{\} \end{cases}$$

tal que $\texttt{n'} = \texttt{n} \oplus \texttt{m}$ para algum $\texttt{n}, \texttt{m} \in \mathbb{Z}$, e onde $\oplus \in \{+, -, *, /, **, \%\}$.

Esta função, garante-nos que o resultado da uma soma, subtracção, multiplicação, divisão, exponenciação ou cálculo do resto da divisão inteira entre duas expressões constantes é ainda uma expressão constate e que o valor inteiro associado ao qualificador da expressão resultante é calculado a partir dos correspondentes valores inteiros das duas expressões íniciais. Se uma destas expressões não for constante o resultado da função será sempre um conjunto vazio.

$$fQual^{const}_{-} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{const}_{-}\ \mathcal{Q}_1\ = \begin{cases} \texttt{const-n} \in \mathcal{Q}_1 & \Rightarrow \{\texttt{const-n'}\} \\ \texttt{const-n} \notin \mathcal{Q}_1 & \Rightarrow \{\} \end{cases}$$

tal que $\texttt{n'} = \texttt{-n}$ para algum $\texttt{n} \in \mathbb{Z}$

Se uma expressão é constante então a sua negação é ainda constante e o valor

associado ao qualificador da expressão resultante é o simétrico do valor associado à expressão inicial.

Operadores booleanos. Como referimos anteriormente, a principal motivação da introdução do qualificador `const` está relacionado com os vectores. Em particular com os parâmetros que definem o tamanho e os intervalos dos *ranges*. Por esta razão neste trabalho apenas destacamos o qualificador `const` sobre inteiros. Assim, todas as funções $fQual_{r_{id}}^{const}$ que actuam sobre as regras de tipagem das operações booleanas devolvem um conjunto vazio.

No entanto, o qualificador `const` poderia também ser utilizado sobre o tipo `bool`, tornando assim possível estabelecer expressões booleanas constantes. O seu funcionamento seria em tudo idêntico ao do qualificador `const` sobre inteiros. Sintaticamente existiria apenas um qualificador, `const` que internamente teria duas representações possíveis: `const-true` e `const-false`. A inferência deste qualificador sobre booleanos seria também idêntica à versão sobre inteiros. As funções $fQual_{r_{id}}^{const}$ definir-se-iam de forma análoga as funções que mostramos anteriormente para o `const` sobre o tipo `int`, mas neste caso, utilizariam operações boleanas para determinar se o (eventual) qualificador devolvido por cada função seria `const-true` ou `const-false`.

Operações sobre vectores. Tal como para os operadores booleanos também o qualificador `const` sobre vectores não será alvo de estudo neste trabalho. Este foca-se apenas no `const` que actua sobre as expressões (inteiras) que acedem aos vectores e não no qualificador `const` sobre os vectores. Assim, as funções $fQual_{@}^{const}$ e $fQual_{[]}^{const}$ devolvem sempre um conjunto vazio.

$$fQual_{@}^{const} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}$$

$$fQual_{@}^{const}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ \{\}$$

$$fQual_{[]}^{const} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}$$

$$fQual_{[]}^{const}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ \{\}$$

Note-se que, na segunda regra, embora a expressão usada para aceder ao vector possa ter o qualificador `const` para que a operação seja correctamente tipada é apenas necessário que o tipo da expressão seja `int`. Nesta situação mesmo que esta expressão não seja constante a operação é correctamente tipada.

No caso dos *ranges*, a regra de tipagem apresentada na Secção 3.6.1 tem que ser alterada de forma a referir explicitamente os qualificadores `const`. Isto porque o valor que estes têm a si associado é utilizado na conclusão da regra para definir o tamanho do vector resultante.

$$\frac{\Gamma \vdash e :: \mathcal{Q}\ \mathsf{Vector}[k]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_1 :: \{\texttt{const-n}\} \cup \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \{\texttt{const-m}\} \cup \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash e[e_1..e_2] :: (fQual\ \mathcal{Q})\ \mathsf{Vector}[n-m+1]\ \mathsf{of}\ \tau}$$

onde, $\texttt{k}, \texttt{n}, \texttt{m} \in \mathbb{Z}$ e $k > m, m \geq n \geq 0$.

Nesta regra, para que o *range* seja tipado correctamente é necessário que ambas as expressões, usadas para definir os limites, tenham um qualificador `const-n`. A função *fQual* é usada para determinar se os restantes qualificadores, que eventualmente ocorram nas expressões utilizadas na regra, são aqui inferidos.

Casts. Ao contrário do que acontece com os qualificadores definidos pelo utilizador, que apresentamos no Capítulo 3, *casts* para `const` não funcionam como um assumir de responsabilidade por parte do programador. Embora este mecanismo possa ser útil (por exemplo para permitir invocação de funções em expressões constantes) é impossível determinar em tempo de compilação o valor que a expressão usada no *cast* tem para assim determinar o valor que o qualificador `const` teria a si associado. Assim, a função $fQual^{const}_{()}$ devolve o qualificador `const` apenas quando a expressão usada no *cast* já tem este qualificador. Ou seja, embora seja possível efectuar *casts* para `const` int estes são redundantes.

$$fQual^{const}_{()} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}$$

$$fQual^{const}_{()}\ \mathcal{Q}_1\ \mathcal{Q}_2 = \begin{cases} \texttt{const-n} \in \mathcal{Q}_1\ \&\&\ \texttt{const} \in \mathcal{Q}_2 & \Rightarrow \{\texttt{const-n}\} \\ \texttt{const-n} \notin \mathcal{Q}_1\ \|\ \texttt{const} \notin \mathcal{Q}_1 & \Rightarrow \{\} \end{cases}$$

para algum $\texttt{n} \in \mathbb{Z}$.

LValues. Recordemos a regra de tipagem das variáveis *LValue* que redefinimos na Secção 4.2.1.

$$\frac{\Gamma(x) = \mathcal{Q}\ \tau}{\Gamma \vdash x :: (fQual\ \mathcal{Q})\ \tau} \quad \tau \in dom(\Gamma),\ \texttt{const} \notin \mathcal{Q}$$

Esta regra tem a restrição $\texttt{const} \notin \mathcal{Q}$ que tem como objectivo impedir que as variáveis constantes possam ocorrer como *LValue*. Na nova versão do qualificador `const` variáveis constantes continuam a não poder ocorrer em *LValues*, portanto, a regra permanece inalterada com excepção desta restrição que tem que ser adaptada aos novos qualificadores `const`. Assim, a restrição $\texttt{const} \notin \mathcal{Q}$ é substituída por $\texttt{const-n} \notin \mathcal{Q}$ para algum $\texttt{n} \in \mathbb{Z}$.

Além disso, a função $fQual^{const}_{var}$ define-se da seguinte forma

$$fQual^{const}_{var} :: \mathcal{Q}_1 \to \mathcal{Q}$$
$$fQual^{const}_{var}\ \mathcal{Q}_1 \ = \ \{\}$$

Esta função devolve sempre um conjunto vazio porque, como $\texttt{const-n} \notin \mathcal{Q}$ então o qualificador nunca poderá ser inferido pela regra.

$$fQual^{const}_{.fi} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}$$
$$fQual^{const}_{.fi}\ \mathcal{Q}_1\ \mathcal{Q}_2 \ = \ \{\}$$

Tal como acontece nos *RValues*, os acessos a campos de *structs* como *LValue* não podem ser utilizados em expressões constantes, assim, a função $fQual^{const}_{.fi}$ devolve sempre um conjunto vazio.

$$fQual^{const}_{[]} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}$$
$$fQual^{const}_{[]}\ \mathcal{Q}_1\ \mathcal{Q}_2 \ = \ \{\}$$

Também nos acessos a vectores, tal como acontece nos *LValues*, a função $fQual^{const}_{[]}$ devolve um conjunto vazio porque estamos apenas a considerar o qualificador `const` sobre vectores.

Tal como nos *ranges RValue*, a regra de tipagem dos *ranges LValue*, apresentada na Secção 3.6.1, tem que ser alterada de forma a referir explicitamente os qualificadores `const`. isto acontece porque o valor que cada `const-n` tem a si associado é utilizado na conclusão da regra para definir o tamanho do vector resultante do range.

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \mathsf{Vector}[k]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_1 :: \{\texttt{const-n}\} \cup \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \{\texttt{const-m}\} \cup \mathcal{Q}_1\ \mathsf{Int}}{\Gamma \vdash l[e_1..e_2] :: (fQual\ \mathcal{Q}_1)\ \mathsf{Vector}[n-m+1]\ \mathsf{of}\ \tau}$$

onde, $\mathtt{k}, \mathtt{n}, \mathtt{m} \in \mathbb{Z}$ e $k > m, m \geq n \geq 0$.

Statements. Como mostramos na Secção 4.2.1 a introdução do qualificador `const` originou alterações nas regras responsáveis pela tipagem das declaração de variáveis, isto para garantir que variáveis `const` não são declaradas sem a correspondente inicialização na mesma instrução. Com a nova versão do qualificador `const`, onde cada qualificador tem a si associado um determinado valor, é necessário realizar algumas alterações adicionais ao sistema de tipos da linguagem. Em particular às regras responsáveis pela declaração com inicialização de variáveis.

Declarações. Relativamente à declaração sem inicialização de variáveis, a regra que apresentamos na Secção 4.2.1 responsável pela tipagem desta operação permanece inalterada. Ou seja, continua a não ser possível declarar variáveis `const` sem a respectiva inicialização. O mesmo acontece com a declaração de funções onde continua a não ser possível utilizar qualquer qualificador `const`. A regra relativa à declaração de novos tipos, apresentada na Secção 3.6.1, também não sofre qualquer alteração.

Declaração e inicialização de variáveis. Relativamente à declaração com inicialização de variáveis, é necessário introduzir restrições adicionais. A regra responsável por esta operação que apresentamos na Secção 4.2.1 é substituída pela seguinte nova regra:

$$\frac{\Gamma \vdash e :: \mathcal{Q}_1\ \tau}{\Gamma \models_\rho \mathsf{def}\ x : \mathcal{Q}_2\ \tau := e :: (\bullet, \Gamma[x :: \mathcal{Q}_2\ \tau])} \quad x \notin dom(\Gamma)$$

onde, $\texttt{const} \in \mathcal{Q}_2 \Rightarrow \exists^1_n \in \mathbb{Z} : \texttt{const-n} \in \mathcal{Q}_1$ e $\mathcal{Q}_2/\{\texttt{const}\} \subseteq \mathcal{Q}_1/\{\texttt{const-n}\}$.

A única alteração esta relacionada com a restrição $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$ que nesta versão do qualificador `const` é substituída por duas restrições. A primeira, $\texttt{const} \in \mathcal{Q}_2 \Rightarrow \exists^1_n \in \mathbb{Z} : \texttt{const-n} \in \mathcal{Q}_1$, obriga a que sempre que uma variável (do tipo `int`) for declarada com o qualificador `const` a expressão usada na atribulação correspondente tem obrigatoriamente que ter a si associado um qualificador `cons-n`, sendo este único. A segunda restrição, $\mathcal{Q}_2/\{\texttt{const}\} \subseteq \mathcal{Q}_1/\{\texttt{const-n}\}$, é análoga à restrição da regra original. O conjunto de qualificadores da variável tem que obrigatoriamente estar contido no conjunto de qualificadores da expressão usada para inicializar a variável.

No entanto, este regra não captura o caso em que a variável declarada é do tipo `vector`. Para permitir que o parâmetro que define o tamanho dos vectores seja uma expressão inteira é necessário diferenciar a declaração de vectores das restantes declarações. Assim, define-se a seguinte regra, que explicitamente evidencia o tipo e conjunto de qualificadores da expressão usada como parâmetro.

$$\frac{\Gamma \vdash e :: \mathcal{Q}' \ \mathsf{int} \quad \Gamma \vdash e_1 :: \mathcal{Q}_1 \ \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau}{\Gamma \models_\rho \mathsf{def}\ x : \mathcal{Q}\ \mathsf{Vector}\ [e]\ \mathsf{of}\ \tau := e_1 :: (\bullet, \Gamma[x :: \mathcal{Q}\ \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau])}$$

onde, $x \notin dom(\Gamma)$ e $\texttt{const-n} \in \mathcal{Q}'$

Nesta nova regra, a restrição $\texttt{const-n} \in \mathcal{Q}'$ obriga a que o parâmetro que define o tamanho do vector seja efectivamente constante. Além disso, o tamanho deste novo vector é igual ao tamanho do vector usado como *RValue*.

Também a regra responsável pela declaração de vectores com inicialização de cada uma das suas posições sofre alterações de forma a permitir que o parâmetro que define o seu tamanho possa ser uma expressão constante. Assim, a regra responsável por esta operação presente na Secção 3.6.1 é substituída pela seguinte regra:

$$\frac{\Gamma \vdash e :: \mathcal{Q}' \ \mathsf{int} \quad \Gamma \vdash e_1 :: \mathcal{Q}_1\ \tau, \ldots, \Gamma \vdash e_n :: \mathcal{Q}_n\ \tau}{\Gamma \models_\rho \mathsf{def}\ x : \mathcal{Q}\ \mathsf{Vector}\ [e]\ \mathsf{of}\ \tau := \{e_1, \ldots, e_n\} :: (\bullet, \Gamma[x :: \mathcal{Q}\ \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau])}$$

onde, $x \notin dom(\Gamma)$ e $\texttt{const-n} \in \mathcal{Q}'$

Tal como no caso anterior, neste regra a restrição $\texttt{const-n} \in \mathcal{Q}'$ possibilita que expressões constantes possam ser utilizadas como parâmetro para definir o tamanho do vector. Além disso, o número de expressões atribuídas ao vector é igual ao valor `n` associado ao qualificador `const-n` da expressão usada como parâmetro.

Assignments. Como referimos anteriormente, variáveis constantes não podem ocorrer como *LValue* em *assignments*. No entanto, as regras responsáveis pela tipagem dos *LValues* já impedem que estes tenham qualquer qualificador `const`, assim, não é necessário realizar essa restrição nas regras dos *assignments*. Desta forma, as regras apresentadas na Secção 3.6.1, relativas aos *assignments*, não sofrem qualquer alteração.

Funções. Na Secção 4.2.1 apresentamos as regras de tipagem das funções adaptadas de forma a impedir que os seus argumentos ou valores de retorno tenham o qualificador `const`. Nesta versão do qualificador estas propriedades mantêm-se, e portanto as regras permanecem inalteradas.

Relativamente às restantes construções da linguagem, `if's`, `whiles` e sequenciação, cujas regras apresentamos na Secção 3.6.1, não existe qualquer influência por parte dos qualificadores `const` seu funcionamento. A conclusão de cada uma dessas regras não depende em momento algum de qualquer qualificador `const`.

4.3.3 Utilização do `const` na Definição de Novos Qualificadores

Ao longo de toda a Secção 4.3 referimos diversas vezes que a principal motivação que teve na origem do qualificador `const` sobre os inteiros do CAO tinha como base permitir que tanto na declaração como nos *ranges* de vectores os parâmetros utilizados pudessem ser definidos através de expressões constante e não apenas através de literais como acontecia até aqui. Além dos vectores, este qualificador sobre os inteiros tem uma segunda utilização que pode ser útil no CAO. A partir do momento em que é possível construir expressões constantes, podemos utilizar estas expressões na definição de novos qualificadores. Nomeadamente nas condições laterais da definição de qualificadores com a cláusula `case`.

Recordemos a definição do qualificador `positive` apresentado na Figura 3.5. A primeira cláusula `case` nesta definição, é utilizada para garantir que qualquer literal maior que zero possa ter a si associado o qualificador `positive`. Esta cláusula pode, na definição do qualificador `positive`, ser substituída pela seguinte nova cláusula.

```
def A: int;
 A, where const<A> && A > 0;
...
```

Desta forma, qualquer variável constante maior zero passa a poder ser utilizada para atribuir valor a qualquer variável declarada com o qualificador `positive`.

Para que esta nova funcionalidade seja implementada correctamente é necessário alterar a função de avaliação definida na Secçao 3.6.2.2. Esta função é responsável pela avaliação das condições laterais das cláusulas usadas da definição dos qualificadores `case`. Mais concretamente, é necessário adaptar a função auxiliar `val` de forma a permitir que está receba como argumento expressões constantes e devolva o valor inteiro associado ao qualificador `const` de cada uma dessas expressões.

Capítulo 5

Utilização dos Qualificadores para não Interferência

5.1 Introdução

Num contexto criptográfico um dos factores fundamentais que devemos ter em conta é a segurança do código que implementa primitivas criptográficas. Aqui, um dos aspectos mais importantes é a análise do fluxo de informação [28] dentro dos programas. essa análise consiste em determinar estaticamente a relação que existe entre os valores retornados e os valores de entrada de um determinado programa e tem dois objectivos fundamentais: impedir que informação sensível seja divulgada por canais não seguros e impedir que informação íntegra dentro dos programas seja corrompida por informação não segura.

Existem vários métodos para efectuar esta análise, desde *theorem proving* [29], interpretação abstracta [30] ou através dos sistemas de tipos [31]. No contexto deste trabalho a análise do fluxo de informação dos programas é assegurada pelo sistemas de tipos. Para tal, são adicionados aos tipos da linguagem etiquetas sob a forma de qualificadores com informação adicional.

Como vimos no Capítulo 3, através de qualificadores de tipos podemos estabelecer diversas restrições sobre os tipos de uma linguagem. Estas restrições podem ser usadas para controlar a informação que circula nos programas. Em particular, podemos utilizar qualificadores de tipos para raciocinar sobre a confiabilidade dessa informação.

O controlo do fluxo de informação incide fundamentalmente em cinco situações concretas dos programas onde a forma como a informação é manipulada influencia em grande medida o seu comportamento. Estas situações estão identificadas e são as seguintes:

- Canais de entrada de informação
- Testes nas estruturas de controlo
- *Assignments*
- Acessos a memória
- Canais de saída de informação

É portanto nestas 5 situações que os qualificadores serão utilizados para controlar o fluxo de informação dos programas CAO.

5.2 Qualificador `untrusted`

Como referimos anteriormente, através de qualificadores podemos estabelecer restrições sobre a informação que circula nos programas e em particular raciocinar sobre a sua confiabilidade. Nesse sentido surge o qualificador `untrusted` que, tal como o `const` (apresentado no Capítulo 4), é um qualificador especial que está predefinido na linguagem.

A principal ideia deste qualificador é marcar determinadas variáveis com uma "flag" que nos indica que estas não são confiáveis e, portanto, não podem ser utilizadas em determinados contextos. Tipicamente as variáveis `untrusted` são passadas como argumento nas funções. No entanto estas podem ser definidas livremente dentro dos próprios programas e as restrições na sua utilização são as mesmas que se verificam para as variáveis `untrusted` passadas como argumento.

5.2.1 Sintaxe

A sintaxe relativa à utilização do qualificador `untrusted` é a mesma que usamos para os restantes qualificadores, definidos pelo utilizador ou pré-definidos (`const`).

Ao contrário do `const`, que está definido apenas sobre o tipo `int`, o qualificador `untrusted` pode ser utilizado com qualquer tipo da linguagem, assim, qualquer variável pode ser declarada como sendo `untrusted`.

```
def f(x: untrusted int): void{
        def y: untrusted int := (untrusted int) 512;
        def z: untrusted int := x;
}
```

A inicialização das varáveis `untrusted` pode ser realizada de duas formas, através de um *assignment* envolvendo uma segunda variável `untrusted` ou através de um *cast*.

5.2.2 Restrições na Utilização do Qualificador untrusted

Como a utilização principal do qualificador `untrusted` está relacionada com *information flow* dos programas a utilização das variáveis `untrusted` nas zonas que assinalamos na Secção 5.1 como sendo aquelas onde a análise ao fluxo de informação é fundamental para evitar que informação integra seja corrompida dentro dos programas está condicionada.

Deste modo, variáveis `untrusted` não podem ser utilizadas como teste em estruturas de controlo nem em acessos a vectores. No *return* de funções podem ser usados desde que o tipo de retorno estabelecido na definição da função seja também ele `untrusted`. Como *input* dos programas, não existe qualquer condicionante à utilização de variáveis `untrusted`. Finalmente, nos *assignments* as variáveis `untrusted` têm, como veremos mais adiante, um tratamento diferente das restantes variáveis.

Na Figura 5.1, linhas 2, 3 e 4 são apresentadas as três situações concretas de código CAO onde variáveis `untrusted` não podem ser utilizadas. Em cada uma destas situações é gerado um erro de compilação a informar a inpossibilidade desta utilização.

```
1        def  f(x: untrusted int, v : vector[64] of int) : untrusted int{
2                v[x] := 0;
3                if (x == 0){ v[0] := 1; }
4                while (x != 0) {
5                  v[i] := v[i] + 1;
6                }
7                return x;
8         }
```

FIGURA 5.1: Situações em que variáveis `untrusted` não podem ser utilizadas.

5.2.3 Contágio Variáveis `untrusted`

Como referimos no início do capítulo, a ideia de utilizar qualificadores no *control flow* de programas é impedir que informação íntegra seja corrompida por informação não segura. Desta forma, para além das situações apresentadas na secção anterior onde as variáveis `untrusted` não podem ser utilizadas, é necessário diferenciar uma outra situação onde estas variáveis têm uma utilização específica. Esta situação está relacionada com os *assignments* dos programas, mais precisamente quando as variáveis `untrusted` ocorrem como *RValue*. Neste caso a "flag" (aqui na forma de qualificador `untrusted`) que indica que a variável não é confiável tem que ser propagado para o *LValue* do *assignment*. Ficando este, a partir deste momento, marcado com esta "flag" e portanto, o sistema de tipos tratará esta variável (*LValue*) como qualquer outra variável `untrusted`.

5.2.4 Estruturas e Vectores `untrusted`

Em determinadas situações a utilização do qualificador `untrusted` tem particularidades que podem passar despercebidas ao programador, e desta forma comprometer o correcto funcionamento dos programas. Exemplo disso são os vectores e as *structs* onde o qualificador `untrusted` tem comportamentos diferentes.

Nos vectores, sempre que uma variável `v` (do tipo `vector`, com tamanho `n`) é `untrusted` todos os seus elementos são abrangidos pelo qualificador. Desta forma tanto `v` como todos os `v[i]` (com $0 \leq$ `i` $<$ `n`) são `untrsuted`. Isto acontece porque não é possível que dentro de um vector existam simultaneamente valores qualificados e não qualificados. Pelo contrário, nas *structs* podemos ter simultaneamente campos `untrusted` (declarados como `untrusted` ou "contagiados") e campos não

`untrusted`. Desta forma a utilização de cada um dos campos é totalmente independente dos restantes. Além disso, a partir do momento que uma um dos campos de uma *struct* é `untrusted` esta é também considerada como tal, e portanto não pode ser utilizada nas situações, apresentadas na Secção 5.1, em que variáveis `untrusted` estão proibidas. Vejamos os seguintes exemplos concretos:

```
def a: untrusted int;
typedef point := struct[
         def X: int;
         def Y: int;
        ];
def s: point;
s.X := a;     1
s.Y := 0;     2

def v: vector[12] of int;
v[0] := a;    3
```

No exemplo 1, a variável `s.x` está a ser contagiada uma vez que a variável `a` é `untrusted`. A partir deste momento a estrutura (variável `s`) passa a não poder ser usada em testes de estruturas de controlo. Já o campo contagiado (`s.x`) deixa de poder ser usado tanto em testes de estruturas de controlo como em acessos a memória. No segundo exemplo não existe qualquer contágio e portanto a variável pode ser utilizada normalmente nas diversas situações. Finalmente, no exemplo 3 existe o contagio de um dos elemento do vector e consequentemente todo o vector passa a ser `untrusted`. Portanto, a variável `v` passa a não poder ser usada em testes de estruturas de controlo e todos os `v[i]` deixam de poder ser usados tanto em testes de estruturas de controlo como em acessos a vectores.

5.3 Sistema de Tipos

Tal como acontece com os restantes qualificadores também com o `untrusted` as principais alterações que a sua intrudução traz à linguagem estão relacionadas com o sistema tipos.

No Capitulo 4 mostramos o mecanismo utilizado para permitir ao sistema de tipos do CAO lidar com expressões constantes. No caso particular do qualificador `const` sobre inteiros, apresentado na Secção 4.3, utilizamos uma abordagem diferente para representar o qualificador sintáctico e os correspondentes qualificadores utilizados pelo sistema de tipos (a cada variável declarada com o qualificador sintáctico `const` corresponde um qualificador `const-n` com o qual o sistema de tipos lida).

Com o qualificador `untrusted` acontece algo similar. Sintacticamente existe um único qualificador (`untrusted`) com o qual os programadores definem as restrições pretendidas sobre as variáveis dos seus programas. Internamente o sistema de tipos lida não com este qualificador mas sim com o seu dual (`trusted`). Assim, sempre que uma variável é declarada sem o qualificador `untrusted` o sistema de tipos sabe que esta é confiável e portanto associa a esta variável o qualificador `trusted`. Analogamente, quando uma variável é declarada como `untrusted` o qualificador `trusted` não lhe é associado. Note-se que, não existe qualquer qualificador sintáctico `trusted`. Por defeito qualquer variável declarada sem o qualificador `untrusted` é considerada pelo sistema de tipos como uma variável `trusted`.

5.4 Inferência do Qualificador `untrusted`

Como referimos anteriormente, numa linguagem com qualificadores de tipos, nativos ou definidos pelo utilizador, é fundamental definir explicitamente as diversas situações em que cada qualificador é inferido pelas construções da linguagem. Assim, e tal como fizemos para os restantes, vamos apresentar as várias situações em que o qualificador `untrusted` é inferido pelas regras de tipagem do CAO. Na prática, e tal como acontece com os qualificadores definidos pelo utilizador, este não é inferido pelas regras de tipagem do CAO mas sim propagado das declarações (sob a forma de `untrusted`) para as expressões (como `trusted`). Desta forma, quando ao longo da resto da secção referirmos que o qualificador `trusted` é inferido este é na verdade propagado até à expressão tipada na regra em causa.

Para tal, vamos apresentar a definição das funções $fQual_{r_{id}}^{trusted}$ responsáveis por determinar se o qualificador `trusted` é inferido por cada uma das regras. Mais uma vez, tal como fizemos para o qualificador `const`, a apresentação destas funções segue a mesma ordem pela qual as regras foram apresentadas na Secção 3.6.1.

Para cada regra r_{id} define-se a correspondente função $fQual^{trusted}_{r_{id}}$. Também aqui, a definição das funções utiliza alguns operadores da teoria de conjuntos, nomeadamente os operadores $\cap$ e $\cup$. O conjunto singular com o qualificador `untrusted` é representado da seguinte forma, {`trusted`}.

Variáveis, chamadas de funções e projecções de estruturas.

$$fQual^{trusted}_{var} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{trusted}_{var}\ \mathcal{Q}_1\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Se uma variável tem em a si associado o qualificador `trusted` então quando usada numa expressão o tipo da sub-expressão por si formada será ainda `trusted`.

$$fQual^{trusted}_{fCall} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual^{trusted}_{fCall}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Nas funções, o único conjunto de qualificadores sobre o qual a função $fQual^{trusted}_{fCall}$ actua é o conjunto que qualifica a tipo de retorno da função, os tipos e respectivos qualificadores dos argumentos não interferem com o tipo de retorno das funções e portanto não são considerados pela função. Assim, se o tipo de retorno de uma função for `trusted`, a invocação desta função numa expressão será ainda `trusted`.

$$fQual^{untrusted}_{.fi} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual^{untrusted}_{.fi}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Como referimos anteriormente, nas *structs* podemos ter simultaneamente campos `trusted` e não `trusted`, além disso, basta que um campo seja não `trusted` para que a *struct* também o seja. No entanto, no acesso a um dos elementos de uma *struct* apenas o conjunto de qualificadores desse campo é necessário para determinar se o tipo da expressão resultante do acesso é ou não `trusted`. Assim, a função $fQual^{trusted}_{.fi}$ devolve o conjunto {`trusted`} sempre que este qualificador está em $\mathcal{Q}_2$, conjunto de qualificadores do campo em questão. Os restantes qualificadores, da *struct* ou dos outros campos não influenciam em momento algum o resultado desta função auxiliar.

Operadores aritméticos.

$$fQual_{\oplus}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{\oplus}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1\ \cap\ \mathcal{Q}_2)$$

onde $\oplus \in \{+, -, *, /, **, \%\}$

Esta função, garante-nos que o resultado da soma, subtracção, multiplicação, divisão, exponenciação ou cálculo do resto da divisão inteira entre duas expressões é `trusted` sempre que estas expressões forem também `trusted`. Se uma não for `trusted` então a expressão resultante também não será.

$$fQual_{-}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual_{-}^{trusted}\ \mathcal{Q}_1\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Sempre que uma expressão é `trusted` a sua negação é também uma expressão `trusted`.

Operadores booleanos. Relativamente às operações boooleana, cujas funções $fQual_{r_{id}}^{trusted}$ apresentaremos em seguida, a inferência do qualificador `trusted` é efectuada de forma semelhante em todas regras. Para que a expressão resultante de uma operação booleana seja `trusted` é necessário que todas as expressões usadas pelo operador sejam também `trusted`. Assim, basta que um dos operandos não seja `trusted` para que a expressão final também não seja.

$$fQual_{\oplus}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{\oplus}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1\ \cap\ \mathcal{Q}_2)$$

onde $\oplus \in \{==, !=\}$

$$fQual_{\oplus}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{\oplus}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1\ \cap\ \mathcal{Q}_2)$$

onde $\oplus \in \{<, \leq, >, \geq\}$

$$fQual_{\oplus}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{\oplus}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1\ \cap\ \mathcal{Q}_2)$$

onde $\oplus \in \{||, \&\&, \hat{}\hat{}\}$

$$fQual_{!}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual_{!}^{trusted}\ \mathcal{Q}_1\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Operadores sobre vectores.

$$fQual_{@}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{@}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1\ \cap\ \mathcal{Q}_1)$$

Na concatenação de vectores é necessário que ambos sejam `trusted` para que o vector resultante também seja. Se um não for `trusted` o resultado da concatenação também não será.

Relativamente ao acesso a vectores, a regra que apresentamos na Secção 3.6.1 tem que ser alterada de forma a impedir que as expressões usadas no acesso sejam `untrusted`. Isto porque, como dissemos na Secção 5.2.2, uma das restrições na utilização de variáveis `untrusted` tem precisamente como objectivo impedir que estas possam ser usadas em acessos a vectores.

A regra responsável por esta operação reescreve-se da seguinte forma:

$$\frac{\Gamma \vdash e_1 :: \mathcal{Q}_1\ \mathsf{Vector}[i]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_2 :: \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash e_1[e_2] :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau}$$

onde, $\texttt{trusted} \in \mathcal{Q}_2$

Com a restrição $\texttt{trusted} \in \mathcal{Q}_2$ temos a garantia que todas as expressões usadas no acesso a vectores são `trusted`. Além disso, a função $fQual_{[]}^{trusted}$ define-se da seguinte forma:

$$fQual_{[]}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{[]}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Sempre que um vector é `trusted` todos os seus elementos também o são, assim o acesso a uma posição do vector devolve ainda um valor `trusted`.

Relativamente aos *ranges* acontece algo similar. As expressões utilizadas para aceder ao vector (limites do *range*) têm que obrigatoriamente ser `trusted`. Assim, a regra responsável por esta operação, que apresentamos na Secção 4.3.2.1, tem que ser alterada de forma a impedir que expressões `untrusted` sejam usadas como limite nos *ranges*. A nova regra responsável pela tipagem dos *ranges* é a seguinte:

$$\frac{\Gamma \vdash e :: \mathcal{Q}\ \mathsf{Vector}[k]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_1 :: \{\texttt{const-n}\} \cup \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \{\texttt{const-m}\} \cup \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash e[e_1..e_2] :: (fQual\ \mathcal{Q})\ \mathsf{Vector}[n-m+1]\ \mathsf{of}\ \tau}$$

onde, $\texttt{k}, \texttt{n}, \texttt{m} \in \mathbb{Z}$, $k > m, m \geq n \geq 0$, $\texttt{trusted} \in \mathcal{Q}_1$ e $\texttt{trusted} \in \mathcal{Q}_2$.

A única alteração relativamente à versão anterior da regra é a restrição $\texttt{trusted} \in \mathcal{Q}_1$ e $\texttt{trusted} \in \mathcal{Q}_2$ que impede que expressões `untrusted` sejam usada como limite dos *range*. A função $fQual^{trusted}_{[..]}$, usada para determinar se o o vector resultante do *range* é ou não `trusted`, define-se da seguinte forma:

$$fQual^{trusted}_{[..]} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}_3 \to \mathcal{Q}$$

$$fQual^{trusted}_{[..]}\ \mathcal{Q}_1\ \mathcal{Q}_2\ \mathcal{Q}_3\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Tal como no acesso a vectores, nos *ranges* o resultado da operação é `trusted` apenas quando o vector inicial é também `trusted`.

Casts. No Capítulo 3 mostramos o *cast* como um operador que funcionam como um assumir de responsabilidade por parte do programador. Estes *casts* permitem conversões para tipos qualificados com qualificadores definidos pelo utilizador. Com o qualificador `untrusted` acontece algo similar. É possível fazer *casts* para `untrusted`, mas ao contrário dos qualificadores definidos pelo utilizador, neste caso o *cast* não funciona com um assumir de responsabilidade mas sim como uma medida preventiva por parte do programador.

Ao realizarmos um *cast* para `untrusted`, estamos a informar o sistema de tipos que a variável em questão não é, por alguma razão, confiável mesmo que expressão

usada no *cast* não seja `untrusted`. Estes *casts* podem ser úteis, por exemplo, para inicializar variáveis `untrusted`.

A função $fQual_{()}^{trusted}$ utilizada pelo sistema de tipos para determinar se o qualificador `trusted` é inferido pela regra do *cast* definida na Secção 3.6.1 é a seguinte:

$$fQual_{()}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{()}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2 = \begin{cases} \texttt{untrusted} \in \mathcal{Q}_2 & \Rightarrow \{\} \\ \texttt{untrusted} \notin \mathcal{Q}_2 & \Rightarrow (\{\texttt{trusted}\} \cap \mathcal{Q}_1) \end{cases}$$

A função $fQual_{()}^{trusted}$ devolve um conjunto vazio sepre que estamos a fazer um *cast* para `untrusted`, nestes casos `trusted` não é inferido pela regra. Se o *cast* não é para `untrusted` então a expressão final (lado esquerdo do *cast*) será `trusted` apenas quando a expressão que lhe é atribuída (lado direito) for também `trusted`.

LValues.

$$fQual_{var}^{trusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}$$

$$fQual_{var}^{trusted}\ \mathcal{Q}_1 = (\{\texttt{trusted}\} \cap \mathcal{Q}_1)$$

Esta função é analoga à versão sobre *RValues* que apresentamos anteriormente Se uma variável tem em a si associado o qualificador `trusted` então sempre que ocorra como LValue será ainda `trusted`.

$$fQual_{.fi}^{untrusted} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}$$

$$fQual_{.fi}^{untrusted}\ \mathcal{Q}_1\ \mathcal{Q}_2 = (\{\texttt{trusted}\} \cap \mathcal{Q}_1)$$

Também a função responsável pelos acessos a campos de structs é análoga à correspondente versão sobre *RValues*. Nas *structs* podemos ter simultaneamente campos `trusted` e não `trusted`, além disso, basta que um campo seja não `trusted` para que a *struct* também o seja. No entanto, no acesso a um dos elementos de uma *struct* apenas o conjunto de qualificadores desse campo é necessário para determinar se o tipo da expressão resultante do acesso é ou não `trusted`. Assim, a função $fQual_{.fi}^{trusted}$ devolve o conjunto $\{\texttt{trusted}\}$ sempre que este qualificador está em $\mathcal{Q}_2$, conjunto de qualificadores do campo em questão.

Tal como aconteceu com a regra responsável pelos acessos a vectores que ocorrem como *RValue* também a regra responsável pelos acessos a vectores *LValues* tem que ser reescrita de forma a impedir que as expressões usadas nos acessos sejam `untrusted`. Assim a regra responsável por esta operação, que apresentamos na Secção 3.6.1, é substituída pela seguinte nova regra:

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \mathsf{Vector}[i]\ \mathsf{of}\ \tau \quad \Gamma \vdash e :: \mathcal{Q}_2\ \mathsf{Int}}{\Gamma \vdash l[e] :: (fQual\ \mathcal{Q}_1\ \mathcal{Q}_2)\ \tau}$$

onde, $\texttt{trusted} \in \mathcal{Q}_2$.

Nesta nova regra a restrição $\texttt{trusted} \in \mathcal{Q}_2$ garante-nos que todas as expressões usadas no acesso a vectores são `trusted`. Além disso, a função $fQual_{[]}^{trusted}$ define-se da seguinte forma:

$$fQual_{[]}^{trusted} :: \mathcal{Q}_1 \to \mathcal{Q}_2 \to \mathcal{Q}$$

$$fQual_{[]}^{trusted}\ \mathcal{Q}_1\ \mathcal{Q}_2\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

Sempre que um vector é `trusted` todos os seus elementos também o são, assim o acesso a uma posição do vector devolve ainda um valor `trusted`.

Também a regra responsável pelos *ranges LValues*, apesentada na Secçao 4.3.2.1, tem que ser reescrita de forma a garantir que as expressões usadas para definir os limites do vector são `trusted`. A nova regra responsável pela tipagem dos *ranges* é a seguinte:

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \mathsf{Vector}[k]\ \mathsf{of}\ \tau \quad \Gamma \vdash e_1 :: \{\texttt{const-n}\} \cup \mathcal{Q}_1\ \mathsf{Int} \quad \Gamma \vdash e_2 :: \{\texttt{const-m}\} \cup \mathcal{Q}_1\ \mathsf{Int}}{\Gamma \vdash l[e_1..e_2] :: (fQual\ \mathcal{Q}_1)\ \mathsf{Vector}[n-m+1]\ \mathsf{of}\ \tau}$$

onde, $\texttt{k}, \texttt{n}, \texttt{m} \in \mathbb{Z}$, $k > m, m \geq n \geq 0$, $\texttt{trusted} \in \mathcal{Q}_1$ e $\texttt{trusted} \in \mathcal{Q}_2$.

A única alteração relativamente à versão anterior da regra é a restrição $\texttt{trusted} \in \mathcal{Q}_1$ e $\texttt{trusted} \in \mathcal{Q}_2$ que impede que expressões `untrusted` sejam usada como limite dos *range*. A função $fQual_{[..]}^{trusted}$, usada para determinar se o o vector resultante do *range* é ou não `trusted`, define-se sa seguinte forma:

$$fQual^{trusted}_{[..]} :: \mathcal{Q}_1 \rightarrow \mathcal{Q}_2 \rightarrow \mathcal{Q}_3 \rightarrow \mathcal{Q}$$

$$fQual^{trusted}_{[..]}\ \mathcal{Q}_1\ \mathcal{Q}_2\ \mathcal{Q}_3\ =\ (\{\texttt{trusted}\}\ \cap\ \mathcal{Q}_1)$$

O vector resultante de um range é `trusted` apenas quando o vector inicial é também `trusted`.

Statements. Como referimos na Secção 5.2 o principal objectivo do qualificador `untrusted` é impedir que informação integra seja corrompida por informação não segura dentro dos programas. Para suportar estas novas características é necessário realizar algumas alterações ao sistema de tipos da linguagem. Algumas destas alterações são garantidas através das regras e das funções $fQual^{trusted}_{r_{id}}$ que apresentamos nos parágrafos anteriores. No entanto é necessário ainda alterar as regras dos *statements* de forma a garantir a totalidades das restrições que se pretenge que o qualificador `untrusted` imponha sobre as construções da linguagem.

Declarações. Relativamente à declaração de variáveis, a regra que apresentamos na Secção 4.2.1 tem que ser alterada de forma a lidar com o qualificador `untrusted`. Desta forma, a nova regra responsavel pela declaração de variáveis é a seguinte:

$$\frac{}{\Gamma \models_{\rho} \mathsf{def}\ x : \mathcal{Q}\ \tau :: (\bullet, \Gamma[x :: \mathcal{Q}'\ \tau])} \quad x \notin dom(\Gamma),\ \texttt{const} \notin \mathcal{Q}$$

onde, $\mathcal{Q}/\{\texttt{untrusted}\} = \mathcal{Q}'/\{\texttt{trusted}\}$ e $\texttt{untrusted} \in \mathcal{Q} \Leftrightarrow \texttt{trusted} \notin \mathcal{Q}'$

Relativamente à versão anterior da regra existe uma alteração fundamental. O conjunto de qualificador com o qual a variável é declarada pode ser diferente do conjunto de qualificador com que esta fica em Γ. No entanto, estes dois conjuntos diferem apenas na eventual presença do qualificador `untrusted`/`trusted`. Esta restrição é garantida pela condição $\mathcal{Q}/\{\texttt{untrusted}\} = \mathcal{Q}'/\{\texttt{trusted}\}$. A segunda restrição adicionada à regra, $\texttt{untrusted} \in \mathcal{Q} \Leftrightarrow \texttt{trusted} \notin \mathcal{Q}'$, é utilizada para garantir duas propriedades. A primeira, obtida pela implicação ($\Rightarrow$), garante-nos que sempre que uma variável é declara com o qualificador `untrusted` então esta nunca terá, em Γ, o qualificador `trusted`. A segunda propriedade, ($\Leftarrow$), garante que se uma variável é declarada sem o qualificador `untrusted` então em Γ esta terá a si associado o qualificador `trusted`.

Na declaração de funções o qualificador `untrusted` pode ocorrer livremente tanto nos argumentos como nos valores de retorno. Também na declaração de novos tipos este qualificador pode ser utilizado normalmente como qualquer outro qualificador (definido pelo utilizador). Assim, as regras responsáveis pela declaração de tipos e pela declaração de funções apresentadas na Secção 3.6.1 não sofrem qualquer alteração.

Declaração e inicialização de variáveis. Tal como a regra da declaração de variáveis também a regra responsável pela declaração com inicialização de variáveis tem que sofrer algumas alterações. Esta nova regra deverá ter em conta dois aspectos. A presença do qualificado `trusted` no conjunto de qualificadores com que a variável fica em Γ depende da eventual presença do qualificador `untrusted` no conjunto de qualificadores com que a variável é declarada. E, para que esta fique com o qualificador `trusted` é ainda necessário verificar se a expressão que lhe é atribuída é ou não uma expressão `trusted`.

A nova regra responsável por esta operação, que substituiu a apresentada na Secção 4.3.2.1, é a seguinte:

$$\frac{\Gamma \vdash e :: \mathcal{Q}_1\ \tau}{\Gamma \models_\rho \mathsf{def}\ x : \mathcal{Q}_2\ \tau := e :: (\bullet, \Gamma[x :: \mathcal{Q}_3\ \tau])} \quad x \notin dom(\Gamma)$$

onde, $\texttt{const} \in \mathcal{Q}_2 \Rightarrow\ \exists^1_n \in \mathbb{Z} : \texttt{const-n} \in \mathcal{Q}_1,\ \mathcal{Q}_2/\{\texttt{const}\} \subseteq \mathcal{Q}_1/\{\texttt{const-n}\},\ \mathcal{Q}_2/\{\texttt{untrusted}\} = \mathcal{Q}_3/\{\texttt{trusted}\}$

e, $\begin{cases} \texttt{untrusted} \in \mathcal{Q}_2 & \Rightarrow \texttt{trusted} \notin \mathcal{Q}_3 \\ \texttt{untrusted} \notin \mathcal{Q}_2 & \Rightarrow (\texttt{trusted} \in \mathcal{Q}_3 \Leftrightarrow \texttt{trusted} \in \mathcal{Q}_1) \end{cases}$

Esta nova regra tem duas restrições relativamente à sua anterior versão. A primeira restrição, $\mathcal{Q}_2/\{\texttt{untrusted}\} = \mathcal{Q}_3/\{\texttt{trusted}\}$, é análoga à restrição que apresentamos na regra da declaração sem inicialização de variáveis e é utilizada para garantir que o conjunto de qualificadores com que a variável é declarada e o conjunto de qualificadores com que esta fica efectivamente em Γ diferem apenas na eventual presença do qualificador `untrusted`/`trusted`. A segunda restrição está dividia em duas partes. A primeira garante que se a variável é declarada como `untrusted` então esta nunca terá, em Γ, a si associado o qualificador `trusted`. A segunda parte obriga a que, além de a variável não ser declarada como `untrusted`, para que esta em Γ tenha o qualificador `trusted` é ainda necessário que a expressão que lhe é atribuída seja também `trusted` (caso contrário a variável sofreria o contágio que explicamos na Secção 5.2.3).

Assignments. Além da declaração com inicialização de varáveis a única situação onde ocorre o contágio de variáveis `untrusted` é nos *assignments*. Por esta razão a regra relativa a esta operação, apresentada na Secção 3.6.1, tem que ser rescrita de forma a lidar com o qualificador `trusted`. A nova regra responsável pelos *assignments* é a seguinte:

$$\frac{\Gamma \vdash l :: \mathcal{Q}_1\ \tau \qquad \Gamma \vdash e :: \mathcal{Q}_2\ \tau}{\Gamma \models_\tau l := e :: (\bullet, \Gamma[l :: \mathcal{Q}_3\ \tau])}$$

onde, $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$, $\mathcal{Q}_1/\{\texttt{trusted}\} = \mathcal{Q}_3/\{\texttt{trusted}\}$ e, $\begin{cases} \texttt{trusted} \notin \mathcal{Q}_1 & \Rightarrow \texttt{trusted} \notin \mathcal{Q}_3 \\ \texttt{trusted} \in \mathcal{Q}_1 & \Rightarrow (\texttt{trusted} \in \mathcal{Q}_3 \Leftrightarrow \texttt{trusted} \in \mathcal{Q}_2) \end{cases}$

Relativamente à regra original esta nova regra tem duas restrições adicionais. A primeira restrição, $\mathcal{Q}_1/\{\texttt{trusted}\} = \mathcal{Q}_3/\{\texttt{trusted}\}$, é usada para garantir que os conjuntos $\mathcal{Q}_1$ e $\mathcal{Q}_3$ são, tirando o qualificador `trusted`, iguais. A segunda restrição está, mais uma vez, dividida em duas partes. A primeira, $\texttt{trusted} \notin \mathcal{Q}_1 \Rightarrow \texttt{trusted} \notin \mathcal{Q}_3$, garante-nos que, se o *LValue* não é `trusted` então após o *assignment* continuará a ser não `trusted`. A segunda, $\texttt{trusted} \in \mathcal{Q}_1 \Rightarrow (\texttt{trusted} \in \mathcal{Q}_3 \Leftrightarrow \texttt{trusted} \in \mathcal{Q}_2)$, garante-nos que se o *LValue* for `truted` então após o *assignment* continua `trusted` apenas se a expressão que lhe é atribuída for também `trusted` (caso contrário a variável sofreria o contágio que explicamos na Secção 5.2.3 e deixaria de ser `trusted`).

Funções. Tal como as regras responsáveis pela declaração de funções também as regram responsáveis pela sua invocação não sofrem qualquer alteração.

If's e whiles. Como referimos anteriormente, uma das situações concretas em que variáveis `untrusted` não podem ser utilizadas em programas CAO está relacionada com os `if's` e com os `whiles`. Nestas duas estruturas de controlo as expressões booleanas utilizadas como teste não podem ser expressões `untrusted`. Por esta razão as regras responsáveis pelos `if's` e pelos `whiles`, apresentadas na Secção 3.6.1, tem que ser alteradas de forma a garantir que todas as expressões usadas como teste têm a si associado o qualificador `trusted`. Em cada umas dessas regras é necessário realizar apenas uma alteração. Cada regra passa a ter uma condição lateral $\texttt{trusted} \in \mathcal{Q}_1$ onde $\mathcal{Q}_1$ é o conjunto de qualificadores da expressão booleana utilizada como teste na estrutura de controlo.

Por exemplo, a regra responsável pelos `whiles` é substituída pela seguinte nova regra.

$$\frac{\Gamma \vdash b :: \mathcal{Q}_1 \ \mathsf{Bool} \quad \Gamma \models_\tau c :: (\rho, \Gamma')}{\Gamma \models_\tau \mathsf{while}\ b\ \{c\}\ :: (\bullet, \Gamma)} \qquad \rho \in \{\tau, \bullet\},\ \texttt{trusted} \in \mathcal{Q}_1$$

Para cada uma das regras dos `if's` acontece algo análogo.

5.4.1 Exemplos de utilização

Nesta secção vamos apresentar um exemplo ilustrativo da utilidade do qualificador `untrusted` num contexto criptográfico.

Na Figura 5.2 podemos ver o código CAO de uma função responsável pelo calculo da exponenciação entre dois valores. Esta função implementa um algoritmo de exponenciação binária que a torna consideravelmente mais eficiente relativamente à versão tradicional da exponenciação.

```
1  def modular_pow(b : signed bits[128],
2   e : untrusted signed bits[1024], m: int) : int{
3
4   def result : int := 1;
5   while( e > 0){
6    if (e[i] == 1) then { result := (result * b) % m; }
7    else {
8     e := e >> 1;
9     b := (b * b) / m;
10   }
11  }
12 return result;
13 }
```

FIGURA 5.2: Exponenciação binária com qualificadores `untrusted` em CAO.

A exponenciação binária é utilizada em diversas aplicação criptográficas, nomeadamente na multiplicação de uma constante por um ponto em sistemas baseados em curvas elípticas [32] ou na operação de decifragem do algoritmo RSA [33].

$$m = c^d \ mod\ n$$

Onde a mensagem m é calculada através da operação da exponenciação entre o criptograma c e a chave privada d.

O funcionamento do algoritmo de exponenciação binária está baseado na análise aos bits do expoente, quando o i-ésimo bit tem o valor `1`, um determinado conjunto

de intrusões é executado, quando este tem o valor `0` é executado um segundo bloco de instruções comum aos dois casos.

O código apresentado na figura 5.2 pode perfeitamente ser utilizado na operação de decifração do RSA, basta para tal invocar a função `modular_pow` com o criptograma que se pretende decifrar (variável `b`) a chave privada (variável `e`) e com o módulo (variável `m`) usado no algoritmo.

Neste caso o qualificador `untrusted` é usado para "proteger" a chave privada (expoente). Ao qualificarmos a chave com o qualificador `untrusted` estamos por exemplo, a proteger a chave de ataques *side channel* [34] que poderiam permitir que um atacante acedesse a alguma informação da chave. Estes ataques podem ser realizados, por exemplo, através de uma análise ao tempo de execução ou do consumo de energia da máquina onde está a ser executada a função. Esta análise tem como objectivo identificar as iterações do ciclo em que uma maior quantidade de energia é consumida ou uma maior quantidade de tempo é dispendida e desta forma perceber se em cada iteração o bit do expoente é `0` ou `1` e assim descobrir a chave privada usada para decifrar o criptograma.

Com o uso do qualficador `untrusted`, e como variáveis `untrusted` não podem ser usadas como teste em estruturas de controlo o expoente não poderia ser usado no teste da linha 6 e portanto esta versão não segura do algoritmo de decifragem RSA não poderia ser compilada em CAO.

Capítulo 6

Conclusão e Trabalho Futuro

6.1 Conclusão

Nesta dissertação apresentamos um conjunto de extensões de alto nível a uma linguagem de domínio específico vocacionada para a implementação de primitivas criptográficas. Estas extensões são baseadas em qualificadores de tipos, que fornecem uma forma leve e simples que permite melhorar a qualidade do software tornando-o mais robusto e menos vulnerável à ocorrência de erros de programação que comprometam o seu bom funcionamento.

O nosso sistema de qualificação, além de fornecer qualificadores pré-definidos na linguagem destinados a situações concretas e portanto cuja utilização é restrita, permite que os utilizadores definam os seus próprios qualificadores. Cada qualificador, pré-definido ou definido pelo utilizador, funciona como uma propriedade atómica sobre os tipos da linguagem. Desta forma, sempre que um utilizador define um novo qualificador sobre um determinado tipo está a refina-lo permitindo assim que este assegure propriedades que tipicamente não são capturadas pelos sistemas de tipos.

Apresentamos duas formas distintas de definir novos qualificadores, cada uma com objectivos distintos. Os qualificadores definidos com a clausula `assume` são utilizados em situações onde o sistema de tipos não pode garantir a consistência da propriedade que o qualificador estabelece. Nestes casos, a responsabilidade da sua utilização é assumida explicitamente pelo programador. Os definidos com

a cláusula `case` permitem discriminar as regras que regem a sua propagação ao conjunto de expressões da linguagem.

Em conclusão, acreditamos que, tanto com os qualificadores nativos como com os definidos pelos utilizadores, quando devidamente utilizados, é possível implementar em CAO uma grande diversidade de primitivas criptográficas com a garantia que muitas das vulnerabilidades relativas à segurança do *software*, que uma implementação em CAO sem qualificadores poderia trazer, são suprimidas nesta versão com qualificadores. Além disso, os qualificadores mostram ser uma mais-valia para os programadores, pois simplificam em grande medida a sua tarefa ao implementar estas primitivas criptográficas.

Finalmente, e apesar de o conjunto de qualificadores actualmente presentes no CAO permitir dar resposta a grande parte dos problemas que este sistema de qualificação pretende resolver, pretende-se que este seja enriquecido, num futuro próximo, com algumas funcionalidades com o objectivo de o tornar ainda mais eficiente.

6.2 Trabalho Futuro

Como referimos na Secção 4.2 existem várias formas de classificar expressões constantes. Actualmente no CAO as expressões constantes são formadas por operações entre literais e variáveis constantes. Na Secção 4.2 referimos ainda que estas expressões constantes podem ser estendidas de forma a permitir que nelas possam ocorrer chamadas a funções. No entanto, para que uma função possa ser usada numa expressão constante tem que necessariamente respeitar um conjunto de requisitos. Esta tem que obrigatoriamente terminar e não pode depender de factores aleatórios nem de variáveis globais não constantes, só assim temos a garantia que a mesma função invocada várias vezes com os mesmos argumentos devolve sempre os mesmos resultados. Além disso, estas funções apenas podem invocar outras funções que cumpram também estes requisitos. A estas funções chamamos funções puras.

Uma forma de classificar as funções que cumprem esta especificação e desta forma diferenciá-las das restantes funções do CAO é através de qualificadores. Aqui em vez de qualificar os tipos, como fizemos ate agora, estamos a qualificar as próprias funções. Assim, definindo-se um qualificador `pure` para as funções, é possível

obrigar a que apenas funções com este qualificador possam ocorrer em expressões constantes e desta forma alargar o leque de expressões constantes do CAO.

Ainda relativamente à noção de constante do CAO outra característica que não está actualmente implementada, e que deixamos como eventual trabalho futuro, tem como objectivo permitir que os parâmetros usados como argumento nas funções possam ser constante. Ao contrário das restantes expressões constantes, as expressões usadas como argumento nas funções não podem ser avaliadas em tempo de compilação e portanto a sua manipulação terá que ser feita sintacticamente. Para tal existem diversas técnicas baseadas em substituição e unificação [35], não estudadas nesta dissertação, que permitem resolver muitos dos problemas que esta utilização das expressões constantes envolve.

Outra característica que actualmente não está implementada no CAO é a possibilidade de se estabelecer uma relação entre os qualificadores e desta forma permitir múltipla qualificação. Como vimos na Secção 3.5.3.1 a sintaxe do CAO já aceita que na definição de cada novo qualificador se construa a relação entre os vários qualificadores, no entanto, este mecanismo não está ainda implementado na linguagem. Além disso, uma relação de ordem entre os qualificadores permite também estender a relação de subtipagem entre os tipos do CAO e desta forma alargar o conjunto de situações em é que possível realizar coerções entre tipos qualificados.

Referências Bibliográficas

[1] Anthony Finkelstein and John Dowell. A comedy of errors: the london ambulance service case study. In *Proceedings of the Eighth International Workshop on Software Specification and Design, IEEE Computer Society Press pp*, pages 2–4. IEEE CS Press, 1996.

[2] Ron Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005. ISBN 0672327988.

[3] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.161279.

[4] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0262032708.

[5] Peter Buchlovsky and Adam Butcher. Buffer overflow vulnerabilities exploits and defensive techniques.

[6] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL*. CEA LIST and INRIA, 2008. Preliminary design (version 1.4, December 12, 2008).

[7] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201700735.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005. ISBN 0321246780.

[10] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 2009. ISBN 0321637631, 9780321637635.

[11] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *SIGPLAN Not.*, 34(5):192–203, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/301631.301665.

[12] P. Ryan, J. McLean, J. Millen, and V. Gligor. Non-interference: Who needs it? *Computer Security Foundations Workshop, IEEE*, 0:0237, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/CSFW.2001.930149.

[13] CACE D5.3. Machine assisted verification and certification tools. Technical Report Deliverable D5.3, CACE Project, 2010.

[14] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics*, pages 157–175, 2001.

[15] Johan Nordlander. Polymorphic subtyping in o'haskell. In *APPSEM Workshop on Subtyping and Dependent Types in Programming, 2000*, 2001.

[16] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[17] J. Turisco D. Joyner, R. Kreminski. *Applied Abstract Algebra*. Johns Hopkins Univ. Press, 2004. URL http://www.usna.edu/Users/math/wdj/book/.

[18] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996. ISBN 0849385237.

[19] Jon Barwise and John Etchemendy. *Language, Proof and Logic*. 1999.

[20] Tim Freeman and Frank Pfenning. Refinement types for ml. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113468.

[21] Tim Freeman. *Refinement types for ML*. PhD thesis, Pittsburgh, PA, USA, 1994.

[22] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Not.*, 33(5):249–257, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/277652.277732.

[23] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1186632.1186635.

[24] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512531.

[25] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. *SIGPLAN Not.*, 40(6):85–95, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1064978.1065022.

[26] Brian Chin, Shane Markstrum, Todd D. Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, pages 264–278, 2006.

[27] Frank Stephan. Set theory, 2010.

[28] Peeter Laud and Fr Informatik. Semantics and program analysis of computationally secure information flow, 2001.

[29] Adam Darvas, Reiner Hahnle, David Sands, and David S. A theorem proving approach to analysis of secure information flow, 2003.

[30] Roberto Barbuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Abstract interpretation of operational semantics for secure information flow. *Inf. Process. Lett.*, 83(2):101–108, 2002.

[31] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *FCT*, pages 365–377, 2005.

[32] Elisabeth Oswald. Introduction to elliptic curve cryptography, 2005.

[33] RSA Labs. Pkcs#1 v2.1: Rsa cryptography standard, 2001.

[34] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *J. Comput. Secur.*, 8(2,3):141–158, 2000. ISSN 0926-227X.

[35] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21:93–124, 1989.