UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

Departamento de Computación

Proxecto de Fin de Carreira de Enxeñaría Informática

Un sistema de tipos Damas-Milner extendido con *predicate subtypes* para el desarrollo de software correcto

Iago Abal

(Tutor: Gilberto Pérez Vega)

iago.abal@udc.es

26 de septiembre de 2012

Mapa de la presentación

Mapa de la presentación

El problema de la confiabilidad del software

- Nuestra sociedad está construida sobre software.
- Los errores software...
  - tienen consecuencias impredecibles,
  - nos afectan a todos,
  - y son muy frecuentes y difíciles de detectar.
- Software confiable.
  - Correcto $\neq$ Confiable.
  - Tony Hoare: «*el software libre de errores es un sueño inalcanzable*».
  - ... como en cualquier otra ingeniería.

Definición (Confiabilidad software)

The probability of failure-free software operation for a specified period of time in a specified environment. — ANSI/IEEE 729-199

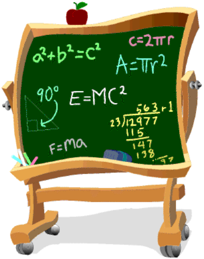
¿Cómo desarrollamos software confiable?

1. Lenguajes de programación con tipado fuerte y estático.
   - Menos propensos a errores.
   - Los lenguajes funcionales son especialmente apropiados.
2. Pruebas de caja blanca/negra.
   - Criterios de cobertura.
   - Generación automática de casos de prueba.
     - E.g. Pex, QuickCheck.
3. Programación por contrato.
   - Precondiciones, postcondiciones e invariantes.
   - Enriquecen/complementan al sistema de tipos.
   - Verificación automática de contratos.
     - E.g. ESC/Java, .NET Code Contracts.

Contribuciones

1. Extensión del sistema de tipos Damas-Milner con *contratos*.
   - Concepto de *contrato* integrado en el sistema de tipos.
   - Contratos polimórficos.
   - Inferencia de contratos.
2. Prueba de concepto: el lenguaje functional $\mathcal{H}_{\text{SPEC}}$.
   - Sintaxis y semántica inspirada en Haskell.
   - Soporta programación por contratos.
   - Verificación de contratos basada en QuickCheck.
   - Aproximación *lightweight* al desarrollo de software correcto.

Teoría de tipos



Sistema de tipos Damas-Milner

- Clasifica expresiones según la clase de valores que computan.
  - E.g. $\vdash 1 : \mathit{int}$, $\vdash$ `'a'` $: \mathit{char}$
- Un *tipo* denota un conjunto de valores.
  - $[\![\mathit{int}]\!] = \mathbb{Z} \equiv \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- Tipado: Prueba la ausencia de errores de tipo.
  - Comprobación: $\Gamma \vdash t : A$
  - Inferencia: $\Gamma \vdash t : A$
- El sistema de tipos Damas-Milner (o Hindley-Milner).
  - Un sistema de tipos polimórfico.
  - Base de los lenguajes funcionales modernos: Caml, Haskell, etc.
  - Polimorfismo implícito.
    - E.g. función identidad: $\mathit{id}\ x = x$
      $\vdash \mathit{id} : \forall a.\ a \to a$
      $\vdash \mathit{id}\ 1 : \mathit{int}$, $\vdash \mathit{id}$ `'a'` $: \mathit{char}$
  - Algoritmo $\mathcal{W}$. (Damas & Milner, 1982)

Predicate subtypes

- Subtipado en base a predicados lógicos.
  - E.g. Números naturales: $nat \mathrel{\hat{=}} \{x : int | x \geq 0\}$
- Concepto inspirado en la interpretación: *tipo* = *conjunto*.
  - $[\![int]\!] = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - Notación de conjuntos: $\{x \in int | x \geq 0\}$
  - En teoría de tipos se conocen como *subset types*.
- El sistema PVS popularizó esta *feature*.
  - Subtipado basado en *type correctness conditions* (TCC).

$$\frac{\Gamma \vdash t : A \qquad \boxed{\vdash_\Gamma P[t]}}{\Gamma \vdash t : \{x : A | P[x]\}}$$

Predicate subtypes

- Subtipado en base a predicados lógicos.
  - E.g. Números naturales: $nat \mathrel{\hat{=}} \{x : int | x \geq 0\}$
- Concepto inspirado en la interpretación: *tipo* = *conjunto*.
  - $[\![int]\!] = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - Notación de conjuntos: $\{x \in int | x \geq 0\}$
  - En teoría de tipos se conocen como *subset types*.
- El sistema PVS popularizó esta *feature*.
  - Subtipado basado en *type correctness conditions* (TCC).

$$\frac{\Gamma \vdash t : \mathbf{int} \qquad \boxed{\vdash_{\Gamma} t \geq 0}}{\Gamma \vdash t : \mathbf{nat}}$$

Predicate subtypes

- Subtipado en base a predicados lógicos.
  - E.g. Números naturales: $nat \mathrel{\hat{=}} \{x : int | x \geq 0\}$
- Concepto inspirado en la interpretación: *tipo* $=$ *conjunto*.
  - $[\![int]\!] = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - Notación de conjuntos: $\{x \in int | x \geq 0\}$
  - En teoría de tipos se conocen como *subset types*.
- El sistema PVS popularizó esta *feature*.
  - Subtipado basado en *type correctness conditions* (TCC).

$$\frac{\Gamma \vdash t : A \qquad \boxed{\vdash_\Gamma P[t]}}{\Gamma \vdash t : \{x : A | P[x]\}}$$

Contratos como tipos

1. Funciones dependientes: $\{x : A\} \rightarrow B$
   - El rango B puede depender del valor de entrada x.
   - E.g. $max : \{a : int\} \rightarrow \{b : int\} \rightarrow \{m : int | m \geq a \wedge m \geq b\}$
2. Tuplas dependientes: $(x : A, B)$
   - La 2ª componente B puede depender del valor de la primera x.
   - E.g. $(a : int, \{b : int | b > a\})$
   - E.g.
     $/ : \{a : real\} \rightarrow \{b : real | b \neq 0\} \rightarrow (q : real, \{r : real | a = q \times b + r\})$

$$\boxed{\text{predicate subtypes} + \text{tipos dependientes} = \textit{contratos}}$$

Mapa de la presentación

head

```
head (x::xs) = x
```

- Dada una lista no-vacia `xs`, `head xs` retorna el primer elemento.
  - Es una función parcial, no está definida para listas vacias.
- En Haskell ...
  - tiene tipo `forall a. [a] -> a`.
  - `head []` causa un error en tiempo de ejecución:
    `*** Exception: Prelude.head: empty list`
- En $\mathcal{H}_{\mathrm{SPEC}}$ ...
  - tiene tipo `forall a.` $\{$`(x::xs):[a]`$\}$ `-> a`.
    - *Pattern subtypes*: subtipos definidos por patrones.
  - `head []` es un término mal tipado:
    `'[]' does not match 'x::x'`
    `In the first argument of 'head', namely '[]'`

Mapa de la presentación

QuickSort

- Algoritmo de ordenación desarrollado por Tony Hoare.
- Caso de estudio típico (y representativo).
- ¿Qué es un algoritmo de ordenación?
- `qsort : {p:[Int]} -> {q:[Int]|sorted q && permutation p q}`
  - El tipo de `qsort` define un algoritmo de ordenación.
  - El sistema de tipos comprueba que la implementación es correcta.
    - Se genera una TCC por cada punto clave de la implementación.
    - Las TCCs pueden ser *testadas* automáticamente.

QuickSort

1. Caso base: la lista vacia ya está ordenada.

```
qsort [] = []
```

2. Sino la lista se subdivide, ordenando y concatenando las sublistas.

```
qsort (x::xs) = qsort ls ++ (x :: qsort rs)
  where (ls,rs) = split x xs
```

QuickSort

1. Caso base: `qsort [] = []`
   - Expresión: `[]`
   - Tipo inferido: `[Int]`
   - Tipo requerido: {`q:[Int]|sorted q && permutation [] q`}
   - Fórmula/TCC: `sorted [] && permutation [] []`
   - `Q.E.D.`
2. Caso inductivo: `qsort (x::xs) = qsort ls ++ (x :: qsort rs)`
   - Expresión: `qsort ls ++ (x :: qsort rs)`
   - Tipo inferido: `[Int]`
   - Tipo requerido: {`q:[Int]|sorted q && permutation (x::xs) q`}
   - Fórmula/TCC:

     ```
     forall (x:Int) (xs:[Int]),
         case split x xs of (ls,rs) ->
             let q = qsort ls ++ (x :: qsort rs)
               in sorted q && permutation (x::xs) q
     ```

   - `+++ OK, passed 100 tests.`

QuickSort

- Introducimos un error deliberadamente.
- Cambiamos `qsort (x::xs) = qsort ls ++ (x :: qsort rs)`,
  por `qsort (x::xs) = qsort rs ++ (x :: qsort ls)`.
  - La lista final está en orden inverso.
  - E.g. `qsort [2,1,3] = [3,2,1]`
- La 2ª TCC pasa a ser:

```
forall (x:Int) (xs:[Int]),
    case split x xs of (ls,rs) ->
        let q = qsort rs ++ (x :: qsort ls)
          in sorted q && permutation (x::xs) q
```

- `*** Failed! Falsifiable (after 3 tests):` $x \mapsto 0$, $xs \mapsto [1]$
  - $ls \mapsto []$, $rs \mapsto [1]$, $q \mapsto [1,0]$
  - sorted $q \equiv$ sorted $[1,0] \equiv \perp$

QuickSort

La 2ª TCC probada en Coq ...

```
Proof.
  intros.
  destruct_call quicksort ; simpl.
  destruct_call quicksort ; simpl.
  simpl in *.
  clear quicksort.
  destruct_conjs ; simpl in *.
  destruct_call split as [l' Hsplit].
  destruct l' ; simpl in * ; myinjection.
  destruct Hsplit as [Hs Hlen].
  split.
  assert(sort le (hd :: x0)).
  apply cons_sort.
  assumption.
  apply (InA_InfA (eqA:=eq)) ; auto with *.
  intros.
  destruct (Hs y).
  pose (permut_InA_InA eq_sym (permut_sym H0) H3).
  intuition ; auto with *.
  apply (sort_le_app H1 H3).
  intros a b.
  destruct (Hs a) ; destruct_conjs.
  intros.
  unfold iff in H4, H5 ; destruct_conjs.
  pose (permut_InA_InA eq_sym (permut_sym H2) H6).
  destruct (InA_In H7) as [bhd|btl].
  destruct (H4 i).
  apply (lt_eq H11 (eq_sym bhd)).
  pose (permut_InA_InA eq_sym (permut_sym H0) btl).
  destruct (proj1 (proj2 (Hs b)) i0).
  pose (proj2 (H4 i)).
  apply lt_le_trans with hd ; auto with *.

  apply permut_add_cons_inside.
  apply permut_tran with (l0 ++ l1) ; auto.
  apply permut_app ; auto.
Qed.
```

Mapa de la presentación

Coercion

$$\boxed{\vdash_\Gamma A \overset{t}{\preceq} B}$$

$$
\begin{array}{rcl}
int \overset{e}{\preceq} int & = & \top \\
[\tau] \overset{e}{\preceq} [\upsilon] & = & \texttt{all}\ \pi_{\tau \preceq \upsilon}\ e \\
\{x : \tau | P[x]\} \overset{e}{\preceq} \upsilon & = & P[e] \Rightarrow \tau \overset{e}{\preceq} \upsilon \\
\tau \overset{e}{\preceq} \{y : \upsilon | Q[x]\} & = & \psi \wedge (\psi \Rightarrow Q[e]) \\
 & \& & \psi := \tau \overset{e}{\preceq} \upsilon \\
(x : \tau_1, \tau_2[x]) \overset{e}{\preceq} (y : \upsilon_1, \upsilon_2[y]) & = & \tau_1 \overset{x}{\preceq} \upsilon_1 \wedge \tau_2[e_1] \preceq \upsilon_2[e_1] \\
 & \& & e_1 := proj_1\ e \\
\{x : \tau_1\} \to \tau_2 \overset{f}{\preceq} \{y : \upsilon_1\} \to \upsilon_2[y] & = & \forall x.\ \psi \wedge (\psi \Rightarrow \tau_2 \overset{f\ x}{\preceq} \upsilon_2[x]) \\
 & \& & \psi := \upsilon_1 \preceq \tau_1
\end{array}
$$

- $t \in A \Rightarrow t \in B$?
  - $A \overset{t}{\preceq} A$
- $nat \overset{t}{\preceq} int \equiv \top$
- $int \overset{t}{\preceq} nat \equiv t \geq 0$
  - $int \overset{1}{\preceq} nat$
  - $int \overset{-1}{\not\preceq} nat$

$nat \doteq \{x : int | x \geq 0\}$

Formulación declarativa

$$\boxed{\Gamma \vdash t : A}$$

$$\textsc{Var}\quad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\textsc{Int}\quad \frac{}{\Gamma \vdash i : int}\; i \in \mathbb{Z}$$

$$\textsc{Abs}\quad \frac{\Gamma, x : A \vdash t : B[x]}{\Gamma \vdash \lambda x.\ t : \{x : A\} \to B[x]}$$

$$\textsc{App}\quad \frac{\Gamma \vdash t : \{x : A\} \to B[x] \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[u]}$$

$$\textsc{Coerce}\quad \frac{\Gamma \vdash t : A \qquad \boxed{\vdash_\Gamma A \overset{t}{\preceq} B}}{\Gamma \vdash t : B}$$

Formulación declarativa

$$\dfrac{\dfrac{\vdots}{n : nat \vdash n - 1 : int} \qquad \boxed{\vdash_{n:nat} int \overset{n-1}{\preceq} \{x : int | \mathbf{P}[x]\}}}{n : nat \vdash n - 1 : \{x : int | \mathbf{P}[x]\}} \; \textsc{Coerce}$$

- La derivación **prueba** que $\vdash n - 1 : int$
- $\vdash n - 1 : \{x : int | \mathbf{P}[x]\}$ si y sólo si $\boxed{\vdash \forall n : nat, \; \mathbf{P}[n - 1]}$.
  - $P[x] = x < n \Longrightarrow \vdash \forall n : nat, \; n - 1 < n$
  - $P[x] = x \geq 0 \Longrightarrow \nvdash \forall n : nat, \; n - 1 \geq 0$
    - Contraejemplo: $n \mapsto 0$.

$nat \doteq \{x : int | x \geq 0\}$

Inferencia de tipos

- La formulación declarativa requiere hacer "conjeturas".

$$\textsc{Abs}\ \frac{\Gamma, x : A \vdash t : B[x]}{\Gamma \vdash \lambda x.\ t : \{x : A\} \to B[x]} \qquad \textsc{App}\ \frac{\Gamma \vdash t : \{x : A\} \to B[x] \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[u]}$$

- Al derivar un algoritmo debemos eliminar toda conjetura.
  - Si un tipo se desconoce, se introduce una incógnita "?".
  - Cuando el tipo A de un término t debe coincidir con otro tipo B:
    (a) Se fuerza compatibilidad: $A \leadsto B$.
    (b) Se introduce un cast: $\vdash_\Gamma A \overset{t}{\preceq} B$.
  - El tipado genera un sistema de restricciones de compatibilidad.
    - ★ Unificación de Robinson.
    - ★ E.g. $\{x : int | P\} \leadsto \{y : int | Q\}$
    - ★ E.g. $\{[int] \leadsto [?]\} \Longrightarrow \{? \mapsto int\}$

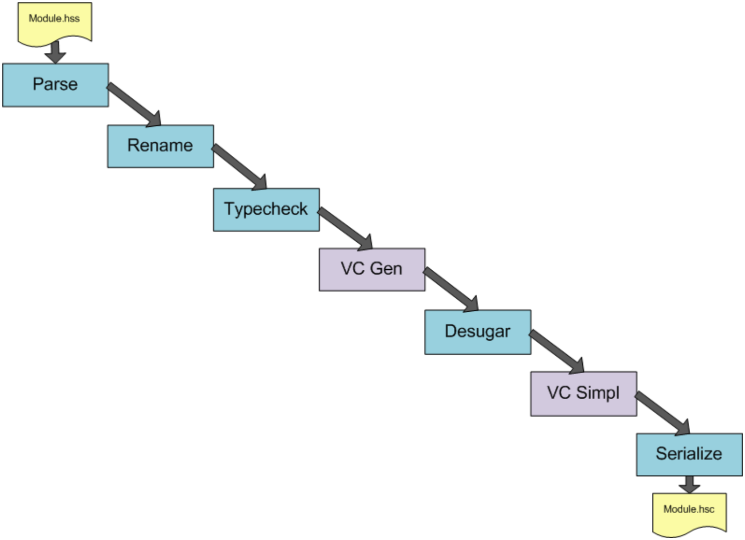
Inferencia de contratos

$$\boxed{A \leadsto B}$$

1. ¿Cómo unificamos predicate subtypes?
   - En lo posible, se propagan los contratos.
   - E.g. $[\{x : int | P\}] \leadsto [?] = \{x : int | P\} \leadsto ? = \{? \mapsto \{x : int | P\}\}$
     - $[\{x : int | P\}] \leadsto [\{x : int | P\}]$ ✓
2. ¿Cómo tratamos tipos dependientes?
   - En $A \leadsto ?$, A podría hacer referencia a variables fuera de alcance.
   - $A \leadsto ? = \{? \mapsto \kappa(\Gamma, A)\}$
   - $\kappa(\Gamma, A)$ elimina de A referencias a variables $x \notin \Gamma$.
     - $\kappa(\Gamma, \{x : int | x \geq 0 \wedge x \geq n\}) = \{x : int | x \geq 0\}$ si $n \notin \Gamma$
3. Inferencia local de tipos. (Pierce & Turner, 1998)
   - Propagación “hacia dentro” de la información de tipos.

Mapa de la presentación

El compilador HSC



El back-end de verificación

1. QuickCheck: *Testing* aleatorio.
   - Prueba $\forall x : A,\ P[x]$ generando x aleatorios con los que *testar* $P[x]$.
     - E.g. $P[0]$, $P[2]$, $P[10]$, $P[31]$, ...
   - Para $\{x : A|P\}$ se generan $x : A$ y se filtran por P.
     - Se reconocen casos especiales como $\{x : int|x \geq k\}$.
   - Soporta la generación de tipos dependientes. ✓
     - E.g. $(x : int, y : int|y > x)$: $(1, 2)$, $(4, 8)$, ...
   - Todavía no soporta la generación de funciones. ×
2. Yices SMT: Lógica de primer orden modulo teorías.
   - Conversión de TCCs a fórmulas SMT.
   - Yices soporta predicate subtypes y tipos dependientes.

Mapa de la presentación

Casos de estudio

	LOC	Coerciones#	*Testables*#
Prelude	120	18	16 (89 %)
QuickSort	25	5	2 (40 %)
Lambda	56	4	4 (100 %)
ListSet	39	11	11 (100 %)

- Filosofía “paga sólo por lo que usas”.
  - En general es fácil eliminar los errores en ejecución.
- Más eficaz que un *framework* de *testing* convencional.
- ∼80 % de las TCCs pueden ser *testadas* automáticamente.

Conclusiones

1. Un sistema Damas-Milner con contratos incorporados.
2. Un algoritmo de inferencia derivado del Algoritmo $\mathcal{M}$.
    - Una variación muy popular del Algoritmo $\mathcal{W}$.
    - No introduce mecanismos de unificación complejos.
    - La inferencia de contratos es limitada,
    - pero se compensa con *inferencia local de tipos*.
3. Soporta *testing* de forma natural.
    - Cobertura de *path* (camino).
    - El *testing* automático tiene limitaciones,
    - y sería interesante poder proporcionar *hints* al back-end.

Trabajo futuro

1. Un dialecto de Haskell para entornos de misión crítica.
   - Haskell es cada vez más usado en esos entornos,
   - principalmente en el sector financiero.
   - Combinar nuestra extensión con las (varias) de Haskell.
2. Mejorar el simplificador de TCCs.
   - El número de TCCs se podría reducir entre un 25 % y un 80 %,
   - aprovechando toda la información contenida en los tipos.
3. Extender el back-end de verificación.
   - Generación de funciones aleatorias.
   - Permitir al programador especificar generadores.
   - Extender el soporte para SMT.
   - Técnicas de análisis estático.
     - Interpretación abstracta.
     - Ejecución simbólica.
     - ...
   - Traducción a probadores de teoremas: PVS, Coq, etc.

¡Gracias!

¿Preguntas?