Un compilador que preserva tipos de seguridad escrito en Haskell

ALBERTO PARDO

Instituto de Computación
Universidad de la República
Montevideo - Uruguay

(junto con Cecilia Manzino, Univ. Nac. de Rosario, Arg.)

CIBSI 2009

Preservación de seguridad por la compilación

$$\forall s, s' \in \mathit{State}.s \cong_L s' \Longrightarrow [\![P]\!]s \cong_L [\![P]\!]s'$$

Preservación de seguridad por la compilación

$$\forall s, s' \in \mathit{State}.s \cong_L s' \Longrightarrow [\![P]\!]s \cong_L [\![P]\!]s'$$

$\Longrightarrow$

$$\forall s, s' \in \mathit{State}.s \cong_L s' \Longrightarrow [\![\mathcal{C}(P)]\!]s \cong_L [\![\mathcal{C}(P)]\!]s'$$

Lenguaje de alto nivel

Expresiones

$$
\begin{array}{lll}
e ::= n & & \textit{– numeral} \\
\quad \mid & x_L & \textit{– variable pública} \\
\quad \mid & x_H & \textit{– variable confidencial} \\
\quad \mid & e + e & \textit{– adición}
\end{array}
$$

Comandos

$$
\begin{array}{lll}
c ::= x_L := e & & \textit{– asignación var. pública} \\
\quad \mid & x_H := e & \textit{– asignación var. confidencial} \\
\quad \mid & \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c\ \textbf{\textit{else}}\ c & \textit{– condicional} \\
\quad \mid & \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c & \textit{– iteración} \\
\quad \mid & c;c & \textit{– secuencia}
\end{array}
$$

Security types: expresiones

$$\vdash n : \mathit{low} \qquad \vdash x_L : \mathit{low} \qquad \vdash x_H : \mathit{high}$$

$$\frac{\vdash e_1 : st_1 \qquad \vdash e_2 : st_2}{\vdash e_1 + e_2 : \mathbf{max}(st_1, st_2)}$$

Security types: expresiones

$$\vdash n : \mathit{low} \qquad \vdash x_L : \mathit{low} \qquad \vdash x_H : \mathit{high}$$

$$\frac{\vdash e_1 : st_1 \qquad \vdash e_2 : st_2}{\vdash e_1 + e_2 : \mathbf{max}(st_1, st_2)}$$

```
data Exp st where
 IntVal :: Int -> Exp Low
 VarL   :: RefL -> Exp Low
 VarH   :: RefH -> Exp High
 Add    :: Exp st1 -> Exp st2 -> Exp (Max st1 st2)
```

Security types

```
data Low
data High

type family Max n m
type instance Max Low  x = x
type instance Max High x = High

type family Min n m
type instance Min Low  x = Low
type instance Min High x = x
```

Security types: comandos

$$\frac{\vdash e : low}{low \vdash x_L := e} \qquad pc \vdash x_H := e$$

$$\frac{\vdash e : st \qquad pc_1 \vdash c_1 \qquad pc_2 \vdash c_2 \qquad st \leq pc_1 \qquad st \leq pc_2}{\mathbf{min}(pc_1, pc_2) \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2}$$

$$\frac{\vdash e : st \qquad pc \vdash c \qquad st \leq pc}{pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c}$$

$$\frac{pc_1 \vdash c_1 \qquad pc_2 \vdash c_2}{\mathbf{min}(pc_1, pc_2) \vdash c_1 ; c_2}$$

Security types: comandos

```
data Com pc  where
 AssL  :: RefL -> Exp Low -> Com Low
 AssH  :: RefH -> Exp st -> Com High
 If0   :: Exp st -> Com pc1 -> Com pc2
                 -> LEq st (Min pc1 pc2)
                 -> Com (Min pc1 pc2)
 While :: Exp st -> Com pc -> LEq st pc -> Com pc
 Seq   :: Com pc1 -> Com pc2 -> Com (Min pc1 pc2)
```

```
data LEq st st' where
   L1 :: LEq Low x
   L2 :: LEq High High
```

Lenguaje de bajo nivel

Instrucciones

$$
\begin{aligned}
c ::= \; & \textbf{\textit{push}}\ n \\
\mid \; & \textbf{\textit{add}} \\
\mid \; & \textbf{fetch}_{\textbf{L}}\ x_L \\
\mid \; & \textbf{fetch}_{\textbf{H}}\ x_H \\
\mid \; & \textbf{store}_{\textbf{L}}\ x_L \\
\mid \; & \textbf{store}_{\textbf{H}}\ x_H \\
\mid \; & \textbf{\textit{branch}}(c, c) \\
\mid \; & \textbf{\textit{loop}}(c, c) \\
\mid \; & c;c \\
\mid \; & \textbf{\textit{noop}}
\end{aligned}
$$

Semántica operacional

$$\langle c, e, s \rangle \triangleright \langle c', e', s' \rangle$$

$$\langle c, e, s \rangle \triangleright (e', s')$$

Security types

$$ls \vdash \textbf{push}\ n : pc \rightsquigarrow low :: ls$$

$$ls \vdash \textbf{fetch}_{\textbf{L}}\ x_L : pc \rightsquigarrow low :: ls$$

$$ls \vdash \textbf{fetch}_{\textbf{H}}\ x_H : pc \rightsquigarrow high :: ls$$

$$low :: ls \vdash \textbf{store}_{\textbf{L}}\ x_L : low \rightsquigarrow ls$$

$$pc_1 :: ls \vdash \textbf{store}_{\textbf{H}}\ x_H : pc \rightsquigarrow ls$$

$$pc_1 :: pc_2 :: ls \vdash \textbf{add} : pc \rightsquigarrow \textbf{max}(pc_1, pc_2) :: ls$$

$$\frac{ls \vdash c_1 : pc' \rightsquigarrow ls' \qquad ls' \vdash c_2 : pc'' \rightsquigarrow ls''}{ls \vdash c_1 ; c_2 : \textbf{min}(pc', pc'') \rightsquigarrow ls''}$$

Security types

$$\frac{pc \leq pc' \qquad ls \vdash c_1 : pc_1 \rightsquigarrow ls \qquad ls \vdash c_2 : pc_2 \rightsquigarrow ls}{pc :: ls \vdash \mathbf{branch}(c_1, c_2) : \mathbf{min}(pc_1, pc_2) \rightsquigarrow ls}$$

$$\frac{ls \vdash c_1 : pc' \rightsquigarrow pc_1 :: ls' \qquad ls' \vdash c_2 : pc'' \rightsquigarrow ls' \qquad pc_1 \leq pc''}{ls \vdash \mathbf{loop}(c_1, c_2) : \mathbf{min}(pc', pc'') \rightsquigarrow ls'}$$

Security types en Haskell

```
data CodeS env pc env' where
  Push   :: Int  -> CodeS env High (Cons Low env)
  FetchL :: RefL -> CodeS env High (Cons Low env)
  FetchH :: RefH -> CodeS env High (Cons High env)
  StoreL :: RefL -> CodeS (Cons Low env) Low env
  StoreH :: RefH -> CodeS (Cons pc env) High env
  Sum    :: CodeS (Cons pc1 (Cons pc2 env)) High
                  (Cons (Max pc2 pc1) env)
  App    :: CodeS env pc1 env1
         -> CodeS env1 pc2 env2
         -> CodeS env (Min pc1 pc2) env2
  ...
```

Security types en Haskell

```
...
Branch :: CodeS env pc1 env
       -> CodeS env pc2 env
       -> LEq pc (Min pc1 pc2)
       -> CodeS (Cons pc env) (Min pc1 pc2) env
Loop   :: CodeS env pc (Cons pc1 env')
       -> CodeS env' pc2 env'
       -> LEq pc1 pc2
       -> CodeS env (Min pc pc2) env'

data Nil
data Cons pc env
```

El compilador

$$\mathcal{C}[\![n]\!] = \textbf{push}\ n$$
$$\mathcal{C}[\![x_L]\!] = \textbf{fetch}_\textbf{L}\ x_L$$
$$\mathcal{C}[\![x_H]\!] = \textbf{fetch}_\textbf{H}\ x_H$$
$$\mathcal{C}[\![e_1 + e_2]\!] = \mathcal{C}[\![e_1]\!]\,;\mathcal{C}[\![e_2]\!]\,;\mathit{add}$$
$$\mathcal{C}[\![x_L := e]\!] = \mathcal{C}[\![e]\!]\,;\textbf{store}_\textbf{L}\ x_L$$
$$\mathcal{C}[\![x_H := e]\!] = \mathcal{C}[\![e]\!]\,;\textbf{store}_\textbf{H}\ x_H$$
$$\mathcal{C}[\![\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!] = \mathcal{C}[\![e]\!]\,;\textbf{branch}(\mathcal{C}[\![c_1]\!], \mathcal{C}[\![c_2]\!])$$
$$\mathcal{C}[\![\textbf{while}\ e\ \textbf{do}\ c]\!] = \textbf{loop}(\mathcal{C}[\![e]\!], \mathcal{C}[\![c]\!])$$
$$\mathcal{C}[\![c_1;c_2]\!] = \mathcal{C}[\![c_1]\!];\mathcal{C}[\![c_2]\!]$$

El compilador en Haskell

```
compileExp :: Exp st
           -> CodeS env High (Cons st env)

compileExp (IntVal n)   = Push n
compileExp (VarL var)   = FetchL var
compileExp (VarH var)   = FetchH var
compileExp (Add e1 e2) = App (App (compileExp e1)
                                  (compileExp e2))
                             Sum
```

El compilador en Haskell

```
compiler :: Com pc -> CodeS env pc env
compiler (AssL var e)     = App (compileExp e)
                                (StoreL var)
compiler (AssH var e)     = App (compileExp e)
                                (StoreH var)
compiler (Seq c1 c2)      = App (compiler c1)
                                (compiler c2)
compiler (If0 e c1 c2 p) = App (compileExp e)
                                (Branch (compiler c1
                                         (compiler c2
                                         p)
compiler (While e c p)    = Loop (compileExp e)
                                 (compiler c)
                                 p
```

Semántica operacional

$$\langle \textbf{push}\ n, e, s\rangle \triangleright (\mathcal{N}[\![n]\!] : e,\ s)$$

$$\langle \textbf{add}, z_1 : z_2 : e, s\rangle \triangleright (z_1 + z_2 : e,\ s)$$

$$\langle \textbf{fetch}_\textbf{L}\ x_L, e, s\rangle \triangleright (s\ x_L : e,\ s)$$

$$\langle \textbf{store}_\textbf{L}\ x_L, z : e, s\rangle \triangleright (e, s[x_L \mapsto z])$$

$$\langle \mathit{branch}(S_1, S_2), z : e, s\rangle \triangleright \langle S_1, e, s\rangle \quad \texttt{if}\ \ z = 1$$
$$\langle \mathit{branch}(S_1, S_2), z : e, s\rangle \triangleright \langle S_2, e, s\rangle \quad \texttt{if}\ \ z \neq 1$$

$$\langle \mathit{loop}(S_1, S_2), e, s\rangle \triangleright \langle S_1 ; \mathit{branch}(S_2 ; \mathit{loop}(S_1, S_2), \mathit{noop}), e, s\rangle$$