Motivation: if we compile/optimize a typed program, we are often unable to use the source-level type system to type check the target program.

Examples:

1. After register allocation a given variable is used with different types:
   
   ```
   f(x: int) {
       x <- x == int 7;
       x <- not x
       assert (x: bool) ← post condition
   }
   ```

   We want to check (just like in the source), that operations are used with their appropriate operands:

   ```
   == int    both operands are int and result is bool
   not       operand and result is bool
   ```

   We cannot use a type system based on declarations.

   We need to use a data-flow based type checker.

   or VCGen!
Assume that we have predicates of the form
\[ e : \texttt{z} \] (for now \( e \in \{\texttt{int}, \texttt{bool}\} \))

fun(x : \texttt{int})
\[
\begin{align*}
x & \leftarrow x = \texttt{int} \land 7 \\
x & \leftarrow \texttt{not} \ x \\
\texttt{assert}(x : \texttt{bool})
\end{align*}
\]

\[ VC = x : \texttt{int} \land x == 7 : \texttt{bool} \land \texttt{not}(x == 7) : \texttt{bool} \]

\[ VC = x : \texttt{bool} \land \texttt{not} \ x : \texttt{bool} \]

\[ VC = x : \texttt{bool} \]

The VC for the whole function is
\[ \forall x. \ x : \texttt{int} \Rightarrow x : \texttt{int} \land x == 7 : \texttt{bool} \land \texttt{not}(x == 7) : \texttt{bool} \]

which we can prove using some simple inference rules that model the source-level type rules

\[
\begin{align*}
E_1 : \texttt{int} & \quad E_2 : \texttt{int} \\
\hline
E_1 == \texttt{int} & \quad E_2 : \texttt{bool} \\
\texttt{not} E : \texttt{bool}
\end{align*}
\]
Example 2

In the process of compilation, some source-level abstraction is eliminated, exposing concrete representation details:

- Counter union types:
- Concrete representation

\[ p: \text{int + bool} \]

\[ \begin{align*}
\text{case } p \text{ of} \\
\left\{ \begin{array}{ll}
\text{left}(x) \rightarrow x == \text{i32} 5 \\
\text{right}(x) \rightarrow \text{not } x
\end{array} \right.
\end{align*} \]

\( p \) is a pointer to

\[ \text{tag} \mid \text{data} \]

\[ t \leftarrow *p \]

\[ x \leftarrow *(p+1) \]

\[ \begin{align*}
\text{if } t == 0 \text{ then} \\
\text{res } \leftarrow x == \text{i32} 5
\end{align*} \]

\[ \text{else} \\
\text{res } \leftarrow \text{not } x
\]

\[ \text{assert (res : bool)} \]

\[ \text{VC} t == 0 \Rightarrow x : \text{int} \wedge x == 5 : \text{bool} \]

\[ t \neq 0 \Rightarrow x : \text{bool} \wedge \text{not } x : \text{bool} \]

- We cannot assign a source-level type to \( x \) before the conditional.
- But a data-flow boxed approach can keep track of conditionals.
- VCGen can do that too.
Example 3

In the process of optimization the obvious mapping from concrete representation is lost

- bounds checks removed
- tag checking removed
- many other optimizations

How to prove that \(3 - 2 : \text{bool}\)?

Cannot use syntactic rules anymore.

\[
\frac{\text{e} : \tau}{\text{e} = \text{e}' \quad \text{congruence helps us do semantic type checking}} \quad \frac{\text{e}' : \tau}{\text{e}'}
\]

Let's design a system of inference rules for formulas of the kind \(\text{e} : \tau\).

We will use the following language of types

\[
\tau ::= \text{int} \mid \text{bool} \mid \tau \times \tau [\tau] \mid \text{const} (\text{e})
\]

- points to structured type
- array of length \(\tau\)
- equal to \(\text{e}\)

\[
\sigma ::= \tau \mid \sigma_1 \times \sigma_2 \mid \sigma [\tau] \mid \text{int} \mid \tau \mid \tau_1 + \tau_2
\]

- arrays of unspecified length
- type variable
We will want to maintain a few invariants such as

- only 0 and 1 are valid representations for booleans
- if we have a \( \ast \) then we can read and write \((a \ast)\) to that address.

\[
\begin{array}{ll}
0 &: \text{bool} \\
1 &: \text{bool} \\
\end{array}
\]

\[
e : \ast \quad e = e' \\
e' : \ast
\]

We try the rule

\[
e : \ast \\
\frac{}{\text{sel}(m, e) : \ast}
\]

but this is not sound!!

It would allow us to prove

\[
\text{sel}(\text{upd}(m, e, 0), e) : \ast \quad \text{from } e : \ast
\]

from which we can prove \( 0 : \ast \)!!

A big hole in the type system

The congruence rule gives us power but it destroys any abstraction!! Because of that we have to be very explicit about assumptions.

We have to add an assumption that the contents of memory matches the types.

\[
\text{typed}(m)
\]
We need first to define the semantics of typing predicates

- We define a representation function \( R \) that associates with each base type \( \tau \) a set of values in \( U \) that are valid representations of expressions of type \( \tau \). 
- \( U \) is the universe of all base values (machine words in assembly language).

Not all \( R \) are good. We define a set of conditions that each \( R \) must satisfy.

a) \( R(\text{int}) = U \) \( \iff \) all values are valid integers.

b) \( R(\text{bool}) = \{0, 1\} \)

c) \( \forall \in R(\tau_1 \times \tau_2) \rightarrow \forall \in R(\tau_1) \land \forall \in R(\tau_2) \land \\
\forall v \in \tau \rightarrow v + |\tau_1| \in R(\tau_2) \)

We define the size of a structured type

- \(|\tau| = 1 \quad |\tau_1 \times \tau_2| = |\tau_1| + |\tau_2| \)
- \(|\tau[\ell]| = 2 \quad |\tau[\ell]| = |\tau| \quad |\tau_1 + \tau_2| = 1 + \max(|\tau_1|, |\tau_2|) \)
- \(1.1 \) is not defined for type variables. All \( t \) must occur inside a \( \tau^* \) or a \( \tau[l] \).
d) $R((\mathfrak{s}_1 + \mathfrak{s}_2)*) = R((\text{const}(0) \times \mathfrak{s}_1)*) \cup R((\text{const}(1) \times \mathfrak{s}_2)*)$

e) $R(\text{const}(n)) = \{ n \}$

f) $v \in R(\sigma[l])$ then $\forall i \ 0 \leq i < l \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ v + i \cdot |\sigma| \in R(\sigma^*)$

g) $R(\mu t.\sigma)* = R((\mu t.\sigma/\ell_0)^*)$

h) $v \in R(\delta[\ell])$ then $v \in R(\omega^*)$ for any $\omega$

i) $v \in R(\omega^*)$ and $v \in R(\omega'\ast)$ then $\omega = \omega'$

This classifies all memory locations uniquely.

And the meaning of $\text{typed}(m)$ in a given representation:

a) $v \in R(\omega^*)$ then $v \in \text{Dom}(m)$ and $m(v) \in R(\sigma)$

b) $v \in R(\delta[\ell])^*$ then $v, v+1 \in \text{Dom}(m)$ and

$m(v) > 0$ and

$m(v+1) \in R(\delta[m(v)])$
Now we can give the inference rules

\[
\begin{align*}
A & : \sigma[L] \quad I \geq 0 \quad I < L \quad |\sigma| = S \\
S * I + A & : \sigma^* \\
A & : (\sigma_1 \times \sigma_2)^* \quad |\sigma_1| = S_1 \\
S_1 + A & : \sigma_2^* \\
A & : (\sigma_1 \times \sigma_2)^* \\
A & : \sigma_1^* \\
A & : \sigma^* \quad \text{typed}(M) \\
\text{sel}(M, A) & : \sigma \\
E & : \text{const}(E^*) \\
A & : (\sigma[I])^* \quad \text{typed}(M) \\
\text{sel}(M, A + 1) & : \sigma[\text{sel}(M, A)] \\
\emptyset & : \text{bool} \quad \text{true} : \text{bool}
\end{align*}
\]
We can even prove the soundness of these rules. Take the upd rule

\[ \text{Hyp. } R \vdash \text{typed}(M), A \in \mathcal{R}(\alpha^*) \quad E \in \mathcal{R}(\alpha) \]

Need to prove

1) \( \forall \alpha \in \mathcal{R}(\alpha^*) \quad \alpha \in \text{Dom}(m) \quad m'(\alpha) \in \mathcal{R}(\alpha') \)

where \( m' = \text{upd}(m, A, E) \)

   a.1. \( \alpha = \epsilon \) then, by inv. i \( \epsilon = \epsilon' \)

   \[ m'(\epsilon) = E \]

   a.2. \( \alpha \neq \epsilon \) then \( m'(\alpha) = m(\alpha) \)

   \[ (\text{Dom}(m') = \text{Dom}(m)) \]

2) \( \forall \alpha \in \mathcal{R}((\alpha[[]])^*) \) then \( \alpha, \alpha + 1 \in \text{Dom}(m) \)

   \[ m(\alpha) > 0 \]

   \[ m'(\alpha + 1) \in \mathcal{R}((\alpha[m(\alpha)])^*) \]

but \( \alpha \neq E \) \( \Rightarrow m'(\alpha) = m(\alpha) > 0 \)

Now \( \alpha + 1 \in \mathcal{R}((\alpha[m(\alpha)])^*) \) and by a) above we have q.e.d.
A proof procedure for typing predicates
- tactic based, reversing the inference rules
- cannot just do backwards chaining
  because there are vors in hyp. that do not appear in conclusion.

We need a function that given A it finds out all potential Γ such that
\[ A : Γ \]

The idea is that pointers can be created in a limited nr. of ways
- \[ A \text{ is } A' + S \]
  - \( S \) is a constant
  - \( \rightarrow \) second element of a pair
  - \( \rightarrow \) index in an array
- \[ A \text{ is } \text{sel}(M, A') \]
  - try to prove that \( A' : 2 \)
- \[ A \text{ is variable} \]
  - we must find the type of \( A \)
    within assumptions.

This is not going to be complete (what if a pointer is computed in a convoluted way
\[
\begin{align*}
&f((\text{int} * x)) \quad * \quad x + \text{alwaysZero}(c) ; \\
&} \\
\end{align*}
\]