Type Safety for MinML

Comparison to \(\lambda\)-calculus Proof

- MinML type soundness proof similar to our earlier proof for soundness with the \(\text{ok}\) relation
  - Harper covers most of the cases in detail
  - Reuse's some lemmas in Chapter 9
- Uses one important new Lemma
  - Canonical forms Lemma

Inversion Theorems

Substitution and Weakening

Lemma 9.2
1. Typing is not affected by “junk” in the symbol table. If \(\Gamma \vdash e : \tau\) and \(\Gamma \vdash \Gamma\), then \(\Gamma \vdash e : \tau\).

2. Substitution for a variable with type \(\tau\) by an expression of the same type doesn’t affect typing. If \(\Gamma [x : \tau] \vdash e' : \tau'\), and \(\Gamma \vdash e : \tau\), then \(\Gamma \vdash (\text{sub} (x, e)) e' : \tau'\).

Canonical Forms

Lemma 10.2 (Canonical Forms)
Suppose that \(\nu : \tau\) is a closed, well-formed value.

1. If \(\nu = \text{true}\), then either \(\nu = \text{true}\) or \(\nu = \text{false}\).

2. If \(\nu = \text{false}\), then \(\nu = \text{false}\) for some \(\nu\).

3. If \(\nu = \text{true} \rightarrow \text{false}\), then \(\nu = \text{false}\) for some \(\nu\).

A value of a given type must have a given form!
How Canonical Forms are Used

• In the proof of progress we have an expression which either steps or is a value.
• If it is a value, because it is well typed we know it must be of a certain form.
• From that fact we know we can take one more step.

Example: Uses of Canonical Forms

(0) Here \(e = \text{if } \ell \text{ then } a \text{ else } b\), with \(\ell : \text{bool}, a : \tau\), and \(b : \tau\). By the base inductive hypothesis, either \(\ell\) is a value, or there exists \(c : \tau\) such that \(\ell \mapsto c\). If \(\ell\) is a value, then we have by the Canonical Forms Lemma, either \(\ell \mapsto \text{true}\) or \(\ell \mapsto \text{false}\). In the former case \(e \mapsto a\), and in the latter \(e \mapsto b\), as required. If \(\ell\) is not a value, then \(e \mapsto e'\), where \(e' = \text{if } \ell \text{ then } a \text{ else } b\), by Rule 8.10.

(1) Here \(e = \text{apply}(\ell, a)\), with \(\ell : \tau \rightarrow \tau\) and \(a : \tau\). By the first inductive hypothesis, either \(\ell\) is a value, or there exists \(c : \tau\) such that \(\ell \mapsto c\). If \(\ell\) is not a value, then \(e \mapsto \text{apply}(\ell, a)\), by Rule 8.20, as required. By the second inductive hypothesis, either \(\ell\) is a value, or there exists \(c' : \tau\) such that \(\ell \mapsto c\). If \(\ell\) is not a value, then \(e \mapsto e'\), where \(e' = \text{apply}(c', a)\), as required. Finally, if both \(\ell\) and \(a\) are values, then by the Canonical Forms Lemma, \(\ell \mapsto \text{true}\) or \(\ell \mapsto \text{false}\), and \(e \mapsto e'\), where \(e' = \left\langle \ell, a, \text{apply}(\ell, a) \right\rangle\), by Rule 8.16.

Stepping Back

Theorem 10.5 (Safety)
If \(e\) is closed and well-typed, then evaluation of \(e\) can only terminate with a value of the same type. In particular, evaluation cannot “get stuck” in an ill-defined state.

What does “ill-defined” really mean?
Formally defined by the \(\mapsto\) relation but you have to inspect the relation to understand what it guarantees.

Extending MinML with Division

\[
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\frac{}{\Gamma \vdash e_1 / e_2 : \text{int}}
\]

Typing rule above does not guarantee that \(e_2\) non-zero.

Option (2) (a.k.a. The Cheat)

- Add a new expression \(\text{error}\) with \(\text{error} : \tau\)
- Add primitive evaluation and propagation rules:
  \[
  \frac{v_1 / 0 \mapsto \text{error}}{\text{apply}(\text{error}, v_2) \mapsto \text{error}}
  \]
  \[
  \frac{v_1 \mapsto \text{error}}{\text{apply}(v_1, \text{error}) \mapsto \text{error}}
  \]

Dealing with Division

Harper suggests two solutions:
1. Change the type system so it can prove statically the denominator is never non-zero
2. Split errors into two classes getting “stuck” and “run-time error”

Doing 1 is impossible to do in general, but maybe possible for interesting number of cases.
Restated Soundness Theorem

Theorem 10.4 (Safety With Errors)
If an expression is well-typed, it can only evaluate to a value or evaluate
to `error`. It cannot "get stuck" in an ill-defined state.

Proof basically the same with small
modifications mostly to progress which
must take care of `error` cases

Safety with Errors or Cheats?
• Aren’t we cheating by defining away problems
we can’t deal with?
• Why is type-soundness so important
  – What’s wrong with uni-typed languages!
  – Lisp, Python, Perl, ....
• Other advantages of strongly typed languages
  – Canonical forms lemma useful for optimizing
    compilers
  – Can guarantee data abstraction
    – Checkable documentation!

Dealing with Division the Hard Way
• Because of the halting problem we cannot
  in general deal with the division by zero
  problem with just a static-type system
• However, we can use the static-type
  system to minimize and reduce the
  amount of dynamic checking needed
• Will sketch a solution that "works"

Non-Zero Integers
• Introduce a new type `nz_int` represents
  non-zero integers
• Overload or introduce new arithmetic
  operators that track non-zeroness
  \[
  \frac{\text{\( n \neq 0 \)}}{\Gamma \vdash \text{\( n \): int}} \quad \frac{\Gamma \vdash \text{\( e \):: nz_int}}{\Gamma \vdash \text{\( e \):: int}}
  \]
  \[
  \frac{\Gamma \vdash \text{\( e \):: nz_int} \quad \Gamma \vdash \text{\( e_1 \):: nz_int}}{\Gamma \vdash \text{\( e_1 \cdot e_2 \):: nz_int}}
  \]

Non-Zero Integers
• Introduce a dynamic test that refines the
type of normal integers to non-zero
integers

Ruling out Non-Zeroness
Type system lets us track non-zeroness
when we can statically rule out division by
zero errors with this typing rule

\[
\frac{\Gamma \vdash \text{\( e_1 \):: int} \quad \Gamma \vdash \text{\( e_2 \):: nz_int}}{\Gamma \vdash \text{\( e_1 \cdot e_2 \):: int}}
\]
Evaluation of Design

• When we can’t prove non-zeroness we can “cheat” by adding a dynamic test that checks at runtime for non-zeroness
• Is this design better than just adding error?
  – Depends on the extra annotation burden and the value of avoiding the checks

Non-Zero Integers in SML

We can easily implement our non-zero type-system using abstract types and modules in SML

signature NON_ZERO = sig
  type nz_int
  val toInt : nz_int -> int
  val casenz: int -> nz_int option
  val mult : nz_int * nz_int -> nz_int
  val quot : int * nz_int -> int
...

Implementing Signature

• A module implementing our signature must be trusted to be correct
• We use type systems to guarantee that only this trusted module can deal with non-zero integers
• If there is a violation of our non-zero invariant we know it must be in our trusted module

Summary

• Type-checking cannot eliminate all errors
• It can isolate errors to well defined modules
• Lets you avoid too many dynamic checks
• Programmers can leverage the type-system to enforce user defined properties