

Type Error Slicing

What is a type error and how do you locate one?

Christian Haack

DePaul University

`fpl.cs.depaul.edu/chaack`

Joe Wells

Heriot-Watt University

`www.macs.hw.ac.uk/~jwb`

Overview

- **Concepts.**
- Examples.
- Algorithms.
- Completeness and Minimality.
- Conclusion.

Current Type Error Location Reporting

Consider this Standard ML (SML) fragment:

```
fn x => fn y => let val w = x + 1 in w::y end
```

Current Type Error Location Reporting

Consider this Standard ML (SML) fragment:

```
fn x => fn y => let val w = x + 1 in w::y end
```

Suppose an error is made at the highlighted spot:

```
fn x => fn y => let val w = y + 1 in w::y end
```

Current Type Error Location Reporting

Consider this Standard ML (SML) fragment:

```
fn x => fn y => let val w = x + 1 in w::y end
```

Suppose an error is made at the highlighted spot:

```
fn x => fn y => let val w = y + 1 in w::y end
```

Type error reports using the \mathcal{W} algorithm give this location:

```
fn x => fn y => let val w = y + 1 in w::y end
```

Current Type Error Location Reporting

Consider this Standard ML (SML) fragment:

```
fn x => fn y => let val w = x + 1 in w::y end
```

Suppose an error is made at the highlighted spot:

```
fn x => fn y => let val w = y + 1 in w::y end
```

Type error reports using the \mathcal{W} algorithm give this location:

```
fn x => fn y => let val w = y + 1 in w::y end
```

Algorithm \mathcal{M} algorithm gives this location:

```
fn x => fn y => let val w = y + 1 in w::y end
```

Current Type Error Messages are Poor

- Most existing compilers make no effort to accurately locate type errors.

Current Type Error Messages are Poor

- Most existing compilers make no effort to accurately locate type errors.
- They typically report the location where the type checker discovered the error. But this is just one of the locations that jointly cause the error.

Current Type Error Messages are Poor

- Most existing compilers make no effort to accurately locate type errors.
- They typically report the location where the type checker discovered the error. But this is just one of the locations that jointly cause the error.
- As a result, many programmers find type error messages unhelpful, especially for higher-order languages with implicit typing.

Current Type Error Messages are Poor

- Most existing compilers make no effort to accurately locate type errors.
- They typically report the location where the type checker discovered the error. But this is just one of the locations that jointly cause the error.
- As a result, many programmers find type error messages unhelpful, especially for higher-order languages with implicit typing.

For example, a `comp.lang.ml` posting:

“Even though I have some experience with SML/NJ and OCaml, I often find myself *mired* in type errors that take me *forever* to resolve.”

Type Error Slices to the Rescue

For each type error:

- Identify *all* program points that contribute to the error.

Type Error Slices to the Rescue

For each type error:

- Identify *all* program points that contribute to the error.
- Display this *set of program points* or a *program slice* as the error location.

Type Error Slices to the Rescue

For each type error:

- Identify *all* program points that contribute to the error.
- Display this *set of program points* or a *program slice* as the error location.
- The error slice should have the following properties:

Type Error Slices to the Rescue

For each type error:

- Identify *all* program points that contribute to the error.
- Display this *set of program points* or a *program slice* as the error location.
- The error slice should have the following properties:
 - *Completeness*. The error should be explainable independently, just by looking at the slice.

Type Error Slices to the Rescue

For each type error:

- Identify *all* program points that contribute to the error.
- Display this *set of program points* or a *program slice* as the error location.
- The error slice should have the following properties:
 - *Completeness*. The error should be explainable independently, just by looking at the slice.
 - *Minimality*. Every proper subslice should be type-error-free.

Overview

- Concepts.
- **Examples.**
- Algorithms.
- Completeness and Minimality.
- Conclusion.

Example 1

val average = fn weight => fn list =>

```
let val iterator = fn (x,(sum,length)) => (sum + weight x , length+1)
    val (sum,length) = foldl iterator (0,0) list
in sum div length end
```

val find_best = fn weight => fn lists =>

```
let val average = average weight
    val iterator = fn (list,(best,max)) =>
        let val avg_list = average list
        in if avg_list > max then
            (list,avg_list)
        else
            (best,max)
        end
    val (best,_) = foldl iterator (nil,0) lists
in best end
```

val find_best_simple = find_best 1

An Incomplete Error Location

```
val average = fn weight => fn list =>
```

```
  let val iterator = fn (x,(sum,length)) => (sum + weight x , length+1)
      val (sum,length) = foldl iterator (0,0) list
  in sum div length end
```

```
val find_best = fn weight => fn lists =>
```

```
  let val average = average weight
      val iterator = fn (list,(best,max)) =>
        let val avg_list = average list
        in if avg_list > max then
            (list,avg_list)
          else
            (best,max)
        end
      val (best,_) = foldl iterator (nil,0) lists
  in best end
```

```
val find_best_simple = find_best 1
```

Another Incomplete Error Location

```
val average = fn weight => fn list =>
```

```
  let val iterator = fn (x,(sum,length)) => (sum + weight x , length+1)
      val (sum,length) = foldl iterator (0,0) list
  in sum div length end
```

```
val find_best = fn weight => fn lists =>
```

```
  let val average = average weight
      val iterator = fn (list,(best,max)) =>
        let val avg_list = average list
        in if avg_list > max then
            (list,avg_list)
          else
            (best,max)
        end
      val (best,_) = foldl iterator (nil,0) lists
  in best end
```

```
val find_best_simple = find_best 1
```

A Complete and Minimal Error Location

```
val average = fn weight => fn list =>
```

```
  let val iterator = fn (x,(sum,length)) => (sum + weight * x , length+1)
      val (sum,length) = foldl iterator (0,0) list
  in sum div length end
```

```
val find_best = fn weight => fn lists =>
```

```
  let val average = average weight
      val iterator = fn (list,(best,max)) =>
        let val avg_list = average list
        in if avg_list > max then
            (list,avg_list)
          else
            (best,max)
        end
      val (best,_) = foldl iterator (nil,0) lists
  in best end
```

```
val find_best_simple = find_best 1
```

Type Error Slice

type constructor clash,
endpoints: function vs. int

```
(.. val average = fn weight =>  
  (.. weight █ (..) ..)  
  
.. val find_best = fn weight =>  
  (.. average weight ..)  
  
.. find_best 1 ..)
```

A Possible Fix

type constructor clash,
endpoints: function vs. int

```
(.. val average = fn weight =>  
    (.. weight * (..) ..)  
  
.. val find_best = fn weight =>  
    (.. average weight ..)  
  
.. find_best 1 ..)
```

Another Possible Fix

type constructor clash,
endpoints: function vs. int

```
(.. val average = fn weight =>  
  (.. weight | (..) ..)
```

```
.. val find_best = fn weight =>  
  (.. average weight ..)
```

```
.. find_best (fn x => x) ..)
```

Yet Another Possible Fix

type constructor clash,
endpoints: function vs. int

```
(.. val average = fn weight =>  
  (.. weight (..) ..)  
  
.. val find_best = fn weight =>  
  (.. average (fn x => weight * x) ..)  
  
.. find_best 1 ..)
```

Example 2

```
val mapActL = fn iterator => fn (list,state) =>
```

```
  let val iterator' = fn (x,(list,state)) =>
                        let val (x,state) = iterator (x,state)
                        in (list @ x, state) end
  in foldl iterator' (nil,state) list end
```

```
val isEven = fn n => n mod 2 = 0
```

```
val doubleOdds = fn list =>
```

```
  let val iterator = fn (n,inc) => if isEven n then
                                    ( n , inc )
                                    else
                                    ( 2 * n , inc + n )
  in mapActL iterator (list,0) end
```

Overlapping Type Error Location #1

```
val mapActL = fn iterator => fn (list,state) =>
```

```
  let val iterator' = fn (x,(list,state)) =>
    let val (x,state) = iterator (x,state)
    in (list @ x, state) end
  in foldl iterator' (nil,state) list end
```

```
val isEven = fn n => n mod 2 = 0
```

```
val doubleOdds = fn list =>
```

```
  let val iterator = fn (n,inc) => if isEven n then
    ( n , inc )
    else
    ( 2 * n , inc + n )
  in mapActL iterator (list,0) end
```

Overlapping Type Error Location #2

```
val mapActL = fn iterator => fn (list,state) =>
```

```
  let val iterator' = fn (x,(list,state)) =>
    let val (x,state) = iterator (x,state)
    in (list @ x, state) end
  in foldl iterator' (nil,state) list end
```

```
val isEven = fn n => n mod 2 = 0
```

```
val doubleOdds = fn list =>
```

```
  let val iterator = fn (n,inc) => if isEven n then
    ( n , inc )
    else
    ( 2 * n , inc + n )
  in mapActL iterator (list,0) end
```

Fixing Overlapping Slice #1

type constructor clash,
endpoints: `int` vs. `list`

```
(.. val mapActL = fn iterator =>  
  .. val (x, (..)) = iterator (..) .. (..) @ x ..)  
  
.. val iterator = fn (.. n ..) =>  
  if (..) then  
    ( n, (..) )  
  else  
    (.. (..) + n ..)  
  
.. mapActL iterator ..)
```

Fixing Overlapping Slice #1

type constructor clash,
endpoints: `int` vs. `list`

```
(.. val mapActL = fn iterator =>
  .. val (x, (..)) = iterator (..)
  .. (..) @ [ x ] ..)

.. val iterator = fn (.. n ..) =>
  if (..) then
    ( n, (..) )
  else
    (.. (..) + n ..)

.. mapActL iterator ..)
```

Fixing Overlapping Slice #1

type constructor clash,
endpoints: `int` vs. `list`

```
(.. val mapActL = fn iterator =>  
  (.. val (x, (..)) = iterator (..)  
    .. (..) @ x ..)  
  
.. val iterator = fn (.. n ..) =>  
  if (..) then  
    ( [ n ] , (..) )  
  else  
    (.. (..) + n ..)  
  
.. mapActL iterator ..)
```

Fixing Overlapping Type Errors

- In many cases, it is likely that the program must be fixed in the overlap. So perhaps the overlap should be presented in a different color.

Fixing Overlapping Type Errors

- In many cases, it is likely that the program must be fixed in the overlap. So perhaps the overlap should be presented in a different color.
- The rationale is that a single programming mistake can cause several type incompatibilities in other locations.

Fixing Overlapping Type Errors

- In many cases, it is likely that the program must be fixed in the overlap. So perhaps the overlap should be presented in a different color.
- The rationale is that a single programming mistake can cause several type incompatibilities in other locations.
- However, there are cases where this does not hold. For example, when the programmer starts changing a representation and has not finished.

Overview

- Concepts.
- Examples.
- **Algorithms.**
 - **Step 1: Associate a set of type constraints with each program point.**
 - Step 2: Find minimal unsolvable sets of constraints.
 - Step 3: Compute slice representations.
- Completeness and Minimality.
- Conclusion.

Avoiding Hindley/Milner Type System

- No type schemes.
(Polymorphism through universal types. “*System W*”)

Avoiding Hindley/Milner Type System

- No type schemes.
(Polymorphism through universal types. “*System \mathcal{W}* ”)
- Instead, enumerate all use-types of free variables.
(Polymorphism through intersection types. “*System \mathcal{I}* ”)

Avoiding Hindley/Milner Type System

- No type schemes.
(Polymorphism through universal types. “*System \mathcal{W}* ”)
- Instead, enumerate all use-types of free variables.
(Polymorphism through intersection types. “*System \mathcal{T}* ”)
- For SML’s let-polymorphism, System \mathcal{W} and System \mathcal{T} permit the same well-typed closed terms.

Avoiding Hindley/Milner Type System

- No type schemes.
(Polymorphism through universal types. “*System \mathcal{W}* ”)
- Instead, enumerate all use-types of free variables.
(Polymorphism through intersection types. “*System \mathcal{T}* ”)
- For SML’s let-polymorphism, System \mathcal{W} and System \mathcal{T} permit the same well-typed closed terms.
- But the type inference algorithms differ. Algorithm \mathcal{T} is more convenient for accurately tracking error locations.

Generating Type Constraints

$$M \Downarrow \langle \Gamma, ty, C \rangle$$

C is a *set of labeled type constraints*, $ty_1 \stackrel{l}{=} ty_2$

Generating Type Constraints

$$M \Downarrow \langle \Gamma, ty, C \rangle$$

C is a *set of labeled type constraints*, $ty_1 \stackrel{l}{=} ty_2$

Spec:

If $(M \Downarrow \langle \Gamma, ty, C \rangle)$ and s is a solution of C ,
then $\langle s(\Gamma), s(ty) \rangle$ is a *principal typing* of M .

The labels are needed so that unification can track which program location caused each constraint.

Generating Type Constraints

- For each subexpression, generate a fresh type variable that represents the type of this subexpression.

Generating Type Constraints

- For each subexpression, generate a fresh type variable that represents the type of this subexpression.

$$\frac{M \Downarrow \langle \Gamma[x \mapsto S], ty, C \rangle;}{(\lambda x^k.M)^l \Downarrow \langle \Gamma[x \mapsto \{\}], \quad , C_{new} \cup C \rangle}$$

where $C_{new} =$

Generating Type Constraints

- For each subexpression, generate a fresh type variable that represents the type of this subexpression.

$$\frac{M \Downarrow \langle \Gamma[x \mapsto S], ty, C \rangle; \quad a_l \text{ fresh}}{(\lambda x^k.M)^l \Downarrow \langle \Gamma[x \mapsto \{\}], a_l, C_{new} \cup C \rangle}$$

where $C_{new} =$

Generating Type Constraints

- For each subexpression, generate a fresh type variable that represents the type of this subexpression.
- Use type constraints to relate this type variable to the types of its children, and the types of the children to each other.

$$\frac{M \Downarrow \langle \Gamma[x \mapsto S], ty, C \rangle; \quad a_l \text{ fresh}}{(\lambda x^k.M)^l \Downarrow \langle \Gamma[x \mapsto \{\}], a_l, C_{new} \cup C \rangle}$$

where $C_{new} =$

Generating Type Constraints

- For each subexpression, generate a fresh type variable that represents the type of this subexpression.
- Use type constraints to relate this type variable to the types of its children, and the types of the children to each other.

$$\frac{M \Downarrow \langle \Gamma[x \mapsto S], ty, C \rangle; \quad a_k, a_l \text{ fresh}}{(\lambda x^k. M)^l \Downarrow \langle \Gamma[x \mapsto \{\}], a_l, C_{new} \cup C \rangle}$$

where $C_{new} = \{a_k \rightarrow ty \stackrel{l}{=} a_l\}$

Generating Type Constraints

- For each subexpression, generate a fresh type variable that represents the type of this subexpression.
- Use type constraints to relate this type variable to the types of its children, and the types of the children to each other.

$$\frac{M \Downarrow \langle \Gamma[x \mapsto S], ty, C \rangle; \quad a_k, a_l \text{ fresh}}{(\lambda x^k. M)^l \Downarrow \langle \Gamma[x \mapsto \{\}], a_l, C_{new} \cup C \rangle}$$

where $C_{new} = \{a_k \rightarrow ty \stackrel{l}{=} a_l\} \cup \{a_k \stackrel{k}{=} ty_S \mid ty_S \in S\}$

Overview

- Concepts.
- Examples.
- **Algorithms.**
 - Step 1: Associate a set of type constraints with each program point.
 - **Step 2: Find minimal unsolvable sets of constraints.**
 - Step 3: Compute slice representations.
- Completeness and Minimality.
- Conclusion.

Finding a Small Error

Rewrite-based unification where, in addition, the (labels of) constraints that were used in a derivation are recorded.

“History-recording unification”

Finding a Small Error

Rewrite-based unification where, in addition, the (labels of) constraints that were used in a derivation are recorded.

“History-recording unification”

$$\text{unify}(C) = \begin{cases} \text{Success}(\sigma) & \text{if } C \text{ is solvable} \\ \text{Error}(L) & \text{otherwise} \end{cases}$$

where L is set of program point labels, σ type substitution

Finding a Small Error

Rewrite-based unification where, in addition, the (labels of) constraints that were used in a derivation are recorded.

“History-recording unification”

$$\text{unify}(C) = \begin{cases} \text{Success}(\sigma) & \text{if } C \text{ is solvable} \\ \text{Error}(L) & \text{otherwise} \end{cases}$$

where L is set of program point labels, σ type substitution

If $\text{unify}(C) = \text{Error}(L)$, then:

- $\Pi_L(C)$ is unsolvable, ...

Finding a Small Error

Rewrite-based unification where, in addition, the (labels of) constraints that were used in a derivation are recorded.

“History-recording unification”

$$\text{unify}(C) = \begin{cases} \text{Success}(\sigma) & \text{if } C \text{ is solvable} \\ \text{Error}(L) & \text{otherwise} \end{cases}$$

where L is set of program point labels, σ type substitution

If $\text{unify}(C) = \text{Error}(L)$, then:

- $\Pi_L(C)$ is unsolvable, ...
- ... but not necessarily minimally unsolvable.

Finding a Small Error

Rewrite-based unification where, in addition, the (labels of) constraints that were used in a derivation are recorded.

“History-recording unification”

$$\text{unify}(C) = \begin{cases} \text{Success}(\sigma) & \text{if } C \text{ is solvable} \\ \text{Error}(L) & \text{otherwise} \end{cases}$$

where L is set of program point labels, σ type substitution

If $\text{unify}(C) = \text{Error}(L)$, then:

- $\Pi_L(C)$ is unsolvable, ...
- ... but not necessarily minimally unsolvable.
- In our experience, it is usually small (close to minimal).

Finding Minimal Errors

Minimizing a small error:

- Given a small error of size n , one can minimize it by running history-recording unification at most n times on subsets of the small error.

Finding Minimal Errors

Minimizing a small error:

- Given a small error of size n , one can minimize it by running history-recording unification at most n times on subsets of the small error.

Finding several small errors:

- In the worst case, the number of minimal errors grows exponentially in the size of the program.

Finding Minimal Errors

Minimizing a small error:

- Given a small error of size n , one can minimize it by running history-recording unification at most n times on subsets of the small error.

Finding several small errors:

- In the worst case, the number of minimal errors grows exponentially in the size of the program.
- We don't know how to enumerate **all** minimal errors in a realistic time, even if the number of minimal errors is not large.

Finding Minimal Errors

Minimizing a small error:

- Given a small error of size n , one can minimize it by running history-recording unification at most n times on subsets of the small error.

Finding several small errors:

- In the worst case, the number of minimal errors grows exponentially in the size of the program.
- We don't know how to enumerate **all** minimal errors in a realistic time, even if the number of minimal errors is not large.
- We have a simple procedure that finds a few different small errors fast, and then gives up. It runs unification several times on large constraint sets.

Finding Minimal Errors, Summary

- We know how to find several, but not always all, minimal errors in a reasonably efficient way, by repeatedly using a history-recording unification algorithm.

Finding Minimal Errors, Summary

- We know how to find several, but not always all, minimal errors in a reasonably efficient way, by repeatedly using a history-recording unification algorithm.
- There are probably more efficient (but also more complicated) algorithms using unification graphs (Port).

Finding Minimal Errors, Summary

- We know how to find several, but not always all, minimal errors in a reasonably efficient way, by repeatedly using a history-recording unification algorithm.
- There are probably more efficient (but also more complicated) algorithms using unification graphs (Port).
- In the worst case, enumerating *all* minimal errors is infeasible, because there can be an exponential number of them.

Finding Minimal Errors, Summary

- We know how to find several, but not always all, minimal errors in a reasonably efficient way, by repeatedly using a history-recording unification algorithm.
- There are probably more efficient (but also more complicated) algorithms using unification graphs (Port).
- In the worst case, enumerating *all* minimal errors is infeasible, because there can be an exponential number of them.
- SML type inference is worst-case DEXPTIME-complete, but in practice behaves almost linearly, so similarly we hope that this behaves reasonably in practice. Anyway, it is not necessary to enumerate all type error slices.

Overview

- Concepts.
- Examples.
- **Algorithms.**
 - Step 1: Associate a set of type constraints with each program point.
 - Step 2: Find minimal unsolvable sets of constraints.
 - **Step 3: Compute slice representations.**
- Completeness and Minimality.
- Conclusion.

Computing Slice Representation

Extend expression syntax class by:

$$sl \in \text{Slice} \quad ::= \dots \quad | \quad \text{dots}(sl_1, \dots, sl_k) \quad | \quad \dots$$

Computing Slice Representation

Extend expression syntax class by:

$$sl \in \text{Slice} \quad ::= \dots \quad | \quad \text{dots}(sl_1, \dots, sl_k) \quad | \quad \dots$$

For example:

$$\lambda x. \text{dots}(x \text{ dots}(), x + \text{dots}())$$

Computing Slice Representation

Extend expression syntax class by:

$$sl \in \text{Slice} \quad ::= \dots \quad | \quad \text{dots}(sl_1, \dots, sl_k) \quad | \quad \dots$$

For example:

$$\lambda x. \text{dots}(x \text{ dots}(), x + \text{dots}())$$

Textual presentation:

$$\text{fn } x \Rightarrow (\dots \text{ x } (\dots) \dots \text{ x } + (\dots) \dots)$$

Computing Slice Representation

Extend expression syntax class by:

$$sl \in \text{Slice} ::= \dots \mid \text{dots}(sl_1, \dots, sl_k) \mid \dots$$

For example:

$$\lambda x. \text{dots}(x \text{ dots}(), x + \text{dots}())$$

Textual presentation:

$$\text{fn } x \Rightarrow (\dots \text{ x } (\dots) \dots \text{ x } + (\dots) \dots)$$

$\text{slice} : \text{LabelSet} \times \text{Exp} \rightarrow \text{Slice}$

$\text{slice}(L, M)$ replaces all syntax nodes of M that are not contained in L by dots-nodes.

Algorithms, Summary

TypeErrorSlicing =

Slice o FindMinErrors o GenerateConstraints

Overview

- Concepts.
- Examples.
- Algorithms.
- **Completeness and Minimality.**
- Conclusion.

Completeness

Additional typing rule for slices:

$$\frac{\Gamma \vdash sl_i : ty_i \quad \text{for all } i \text{ in } \{1, \dots, k\}}{\Gamma \vdash \text{dots}(sl_1, \dots, sl_k) : ty}$$

Completeness

Additional typing rule for slices:

$$\frac{\Gamma \vdash sl_i : ty_i \quad \text{for all } i \text{ in } \{1, \dots, k\}}{\Gamma \vdash \text{dots}(sl_1, \dots, sl_k) : ty}$$

Theorem (Completeness).

A program slice that is returned by `TypeErrorSlicing` has a type error (i.e. is not typable).

Minimality

Define *subslice ordering* on slices:

$$sl_1 \sqsubset sl_2$$

iff sl_1 is obtained from sl_2 by replacing additional non-dots-nodes by dots-nodes.

Minimality

Define *subslice ordering* on slices:

$$sl_1 \sqsubset sl_2$$

iff sl_1 is obtained from sl_2 by replacing additional non-dots-nodes by dots-nodes.

Theorem (Minimality).

Every proper subslice of a slice sl returned by TypeErrorSlicing has no type error (i.e. is typable),

Minimality

Define *subslice ordering* on slices:

$$sl_1 \sqsubset sl_2$$

iff sl_1 is obtained from sl_2 by replacing additional non-dots-nodes by dots-nodes.

Theorem (Minimality).

Every proper subslice of a slice sl returned by TypeErrorSlicing has no type error (i.e. is typable), provided all bound variables of sl are distinct.

Overview

- Concepts.
- Examples.
- Algorithms.
- Completeness and Minimality.
- **Conclusion.**

Our Contributions

- An explanation of how to represent type error locations as program slices.

Our Contributions

- An explanation of how to represent type error locations as program slices.
- A formal characterization of when a slice is a *complete* and *minimal* representation of a type error.

Our Contributions

- An explanation of how to represent type error locations as program slices.
- A formal characterization of when a slice is a *complete* and *minimal* representation of a type error.
- Simple algorithms for finding complete and minimal type error slices.

Our Contributions

- An explanation of how to represent type error locations as program slices.
- A formal characterization of when a slice is a *complete* and *minimal* representation of a type error.
- Simple algorithms for finding complete and minimal type error slices.
- Proofs that our algorithms yield minimal and complete slices.

Our Contributions

- An explanation of how to represent type error locations as program slices.
- A formal characterization of when a slice is a *complete* and *minimal* representation of a type error.
- Simple algorithms for finding complete and minimal type error slices.
- Proofs that our algorithms yield minimal and complete slices.
- A demonstration implementation available via our web page at `http://www.macs.hw.ac.uk/ultra/compositional-analysis/type-error-slicing/slicing.cgi`.

Related Work

- Directly inspirational:
 - Wand: Locating Sources of Type Errors
 - Dinesh and Tip: Type error slicing for languages with explicit type annotations
 - Yang, Michaelson, Trinder, (Wells): UAE algorithm

Related Work

- Directly inspirational:
 - Wand: Locating Sources of Type Errors
 - Dinesh and Tip: Type error slicing for languages with explicit type annotations
 - Yang, Michaelson, Trinder, (Wells): UAE algorithm
- Recent and similar in spirit:
 - Chopella and Haynes (Indiana, 1995, 2002)
 - Heeren et al. (Utrecht, 2002)
 - Sulzmann et al.: Chameleon type debugger.

Related Work

- Directly inspirational:
 - Wand: Locating Sources of Type Errors
 - Dinesh and Tip: Type error slicing for languages with explicit type annotations
 - Yang, Michaelson, Trinder, (Wells): UAE algorithm
- Recent and similar in spirit:
 - Chopella and Haynes (Indiana, 1995, 2002)
 - Heeren et al. (Utrecht, 2002)
 - Sulzmann et al.: Chameleon type debugger.
- Other work on type errors: Bernstein and Stark, Duggan and Bent, McAdam, Chitil, Flanagan et al. (MrSpidey), etc.

Future Work

- Extend to full SML.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.
 - ...

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.
 - ...
- Add features for interactive debugging.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.
 - ...
- Add features for interactive debugging.
 - Interactive navigation through location sets.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.
 - ...
- Add features for interactive debugging.
 - Interactive navigation through location sets.
 - Animation/rewriting of error slices to help explain them.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.
 - ...
- Add features for interactive debugging.
 - Interactive navigation through location sets.
 - Animation/rewriting of error slices to help explain them.
- More efficient and more compositional algorithms.

Future Work

- Extend to full SML.
 - Type annotations including module signatures.
 - Equality types and overloading.
 - ...
- Add features for interactive debugging.
 - Interactive navigation through location sets.
 - Animation/rewriting of error slices to help explain them.
- More efficient and more compositional algorithms.
- Handle object-oriented languages (both class-based and object-based).