Information Aware Type Systems and Telescopic Constraint Trees

Philippa Cowderoy

MSFP 2020

email: flippa@flippac.org
twitter: @flippac
What I’m Talking About

- Motivating ‘information awareness’
- Tools for information awareness
- Information Aware Type Systems
What I’m Talking About

- Motivating ‘information awareness’
- Tools for information awareness
- Information Aware Type Systems
- Telescopic Constraint Trees
Some people find *AppEq* easier to read than *AppTrad*. Why?

- *AppEq* shows us explicitly the information we infer
- e.g. *Tf* is a function that can take *Tp* as a parameter
What does this \( \tau \) say?
What does this $\tau$ say?

Using schematic variables can mean different things:

- Does it use known information?
- Is it a pattern?
- Is it part of a non-linear pattern?
- Can it be reduced first? (Should it?)
We want an ‘information aware’ approach
Let us be explicit about this!
We want an ‘information aware’ approach
Let us be explicit about this!

Constraints can help us do better...
...But they can’t stop us doing worse
What I’m Talking About Now

- Motivating ‘information awareness’ – DONE
- Tools for information awareness
  - Information Effects
  - Constraints
- Information Aware Type Systems
- Telescopic Constraint Trees
From *Information Effects* by James & Sabry

Mostly-reversible programming
From *Information Effects* by James & Sabry

- Mostly-reversible programming
- Seen via *isomorphic* programming
From *Information Effects* by James & Sabry

- Mostly-reversible programming
- Seen via *isomorphic* programming

- I can still use the idea in just one direction!
Breaches of conservation of information:

- When information is created
- When information is destroyed
Information Effects Are . . .

Breaches of conservation of information:

- When information is created
- When information is destroyed
- When information is duplicated
Breaches of conservation of information:

- When information is created
- When information is destroyed
- When information is duplicated

- Inference creates new information
- Implied by the information we have
Constraints as Information

Syntax

Static Semantics

Dynamic Semantics

Insight: Constraints are a unit of information!
Syntax
- Just what we wrote
- Information as in bits
- Bits are a bad unit for precise accounting!

Static Semantics

Dynamic Semantics
Static Semantics

- What holds if the constraint is satisfied?
- *Satisfaction predicate*
- What shapes of problem are solveable?
Syntax

Static Semantics

Dynamic Semantics

- Part of a *constraint problem*
- Part of the *process* of solving that problem
Contraints as Information

Syntax

Static Semantics

Dynamic Semantics

- Part of the *process* of solving a problem
- A process that can block on unknown information
- . . . Or provide information
- Unknowns represented by *solver variables* or *metavariables*
Constraints as Information

Syntax

Static Semantics

Dynamic Semantics

Insight

Constraints are a unit of information!
What I’m Talking About Now

- Motivating ‘information awareness’ – DONE
- Tools for information awareness – DONE

Information Aware Type Systems
- Aim
- Methods
- Example – Simply Typed Lambda Calculus
- Modes

Telescopic Constraint Trees
Information Aware Type Systems – Aim

- Clear introduction and elimination of information
- Clear data flow
- Clear choices about flow
- In design and implementation
Information Aware Type Systems – Methods

- Linear logic variables: one +ve source, one -ve sink
- Constraints
  - Generation is an information effect
  - Keep dataflow general
  - Flexible abstraction
- Explicit duplication
Constraints for the Simply Typed Lambda Calculus

- $\tau = \tau$ Type equality
- $x : \tau \in \Gamma$ Binding in context
- $\Gamma' := \Gamma ; x : \tau$ Context extension
- $\Gamma \vdash \Gamma$ Context duplication

- Convention: write $L = R$ as if ‘assigning’ to $L$
- Context constraints $\Rightarrow$ structural rules
Type equality
\[ \tau = \tau \]

Binding in context
\[ x : \tau \in \Gamma \]

Context extension
\[ \Gamma' := \Gamma ; x : \tau \]

Context duplication
\[ \Gamma \vdash [\Gamma] \]

Var
\[ \Gamma \vdash x : \tau \]
Type equality

\[ \tau = \tau \]

Binding in context

\[ x : \tau \in \Gamma \]

Context extension

\[ \Gamma' := \Gamma \vdash x : \tau \]

Context duplication

\[ \Gamma \Gamma \]

\[ \Gamma f := \Gamma \vdash x : \tau p \]

\[ \Gamma f \vdash T : \tau r \]

\[ \tau f = \tau p \rightarrow \tau r \]

\[ \Gamma \vdash \lambda x. T : \tau f \text{ Lam} \]
\[ \tau = \tau \quad \text{Type equality} \]
\[ x : \tau \in \Gamma \quad \text{Binding in context} \]
\[ \Gamma' := \Gamma ; x : \tau \quad \text{Context extension} \]
\[ \Gamma \vdash \Gamma \quad \text{Context duplication} \]

\[ \Gamma \vdash \Gamma_f \Gamma_p \]

\[ \Gamma_f \vdash Tf : \tau_f \quad \Gamma_p \vdash Tp : \tau_p \]

\[ \tau_p \rightarrow \tau_r = \tau_f \]

\[ \Gamma \vdash Tf \ Tp : \tau_r \quad \text{App} \]
Modes

- Which way is data flowing?
- When do we know enough to solve a constraint?
Which way is data flowing?
When do we know enough to solve a constraint?

- Data flows +ve to -ve
- Constraints can ‘fire’ when -ve variables known
## Different Modes of a Type System

<table>
<thead>
<tr>
<th>Mode</th>
<th>Unidirectional</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma^+ \vdash T^+ : \tau^+$</td>
<td>Type checking</td>
<td>Checking</td>
</tr>
<tr>
<td>$\Gamma^+ \vdash T^+ : \tau^-$</td>
<td>Free variable types</td>
<td>Synthesis</td>
</tr>
<tr>
<td>$\Gamma^- \vdash T^+ : \tau^+$</td>
<td>Proof search</td>
<td>Checked type</td>
</tr>
<tr>
<td>$\Gamma^- \vdash T^+ : \tau^-$</td>
<td>Program synthesis</td>
<td>Synthesised type</td>
</tr>
</tbody>
</table>

Each mode can have its own implementation (‘procedure’)
The function arrow $\rightarrow$ doesn’t appear in source.

It does appear in our types.

Information we infer from or create about terms.
I assign two different modes to $\rightarrow$:

- Based on how the solver handles $=$ constraints
- LHS of $=$ is being ‘assigned to’ in some form

- $+$ve construction vs -ve pattern-matching
Mode: $\Gamma^+ \vdash T^+ : \tau^-$ (Synthesis or ‘typechecking’)

$x^- : \tau^+ \in \Gamma^-$

$$\Gamma^+ \vdash x^+ : \tau^- \text{ Var}$$
Mode: $\Gamma^+ \vdash T^+ : \tau^-$ (Synthesis or ‘typechecking’)

\[
\Gamma f^+ := \Gamma^- ; x^- : \tau p^+ \\
\Gamma f^- \vdash T^- : \tau r^+ \\
\tau f^+ = \tau p^- \rightarrow^+ \tau r^- \\
\hline \\
\Gamma^+ \vdash \lambda x^+. T^+ : \tau f^- \text{ Lam}
\]
Mode: $\Gamma^+ \vdash T^+ : \tau^-$ (Synthesis or ‘typechecking’)
Information Aware Simply Typed $\lambda$-Calculus

$\Gamma f := \Gamma ; x : \tau p$

$\Gamma f \vdash T : \tau r$

$\tau f = \tau p \rightarrow \tau r$

$\Gamma \vdash x : \tau \hspace{1cm} \text{Var}$

$\Gamma \vdash \lambda x.T : \tau f \hspace{1cm} \text{Lam}$

$\Gamma f \vdash Tf : \tau f$

$\Gamma p \vdash Tp : \tau p$

$\tau p \rightarrow \tau r = \tau f$

$\Gamma \vdash Tf Tp : \tau r \hspace{1cm} \text{App}$
What I’m Talking About Now

- Motivating ‘information awareness’ – DONE
- Tools for information awareness – DONE
- Information Aware Type Systems – DONE

Telescopic Constraint Trees
- What they are and do
- How to build them
- Things to do with them!
What’s a Telescopic Constraint Tree?

A Telescopic Constraint Tree (or TCT) is:

- A Tree – a refined AST
- Containing Constraints
- Telescopic
A Telescopic Constraint Tree (or TCT) is:
- A Tree – a refined AST
- Containing Constraints
  - Also metavariable binders
- Telescopic
What’s a Telescopic Constraint Tree?

A Telescopic Constraint Tree (or TCT) is:

- A Tree – a refined AST
- Containing Constraints
  - Also metavariable binders
- Telescopic
  - Built from composable contexts
What Else is a Telescopic Constraint Tree?

- A typechecking problem in progress
- Generic
- Derivable from Information Aware Type Systems
- A scope-aware constraint store
What Else is a Telescopic Constraint Tree?

- A typechecking problem in progress
- Generic
- Derivable from Information Aware Type Systems
- A scope-aware constraint store
  - Scoping not needed for STLC
  - Useful for HM, dependent types and more!
What Does a Telescopic Constraint Tree Do?

What an ordinary typechecker does
What Does a Telescopic Constraint Tree Do?

What an ordinary typechecker does in time

Telescopic constraint trees do in space
We use these constraints, which refer to contexts:

\[
\begin{align*}
\text{Binding in context} & \quad x : \tau \in \Gamma \\
\text{Context extension} & \quad \Gamma' := \Gamma ; x : \tau
\end{align*}
\]

We can’t put those directly in a telescope: They want to refer to it!
Situated Constraints

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \tau \in \Gamma$</td>
<td>$?x : \tau$</td>
</tr>
<tr>
<td>$\Gamma' := \Gamma ; x : \tau$</td>
<td>$!x : \tau$</td>
</tr>
</tbody>
</table>

The new constraints are *situated*. Their meaning depends on their position in the telescope:

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$?x : \tau$</td>
</tr>
<tr>
<td>Query/Ask for current binding [here]</td>
</tr>
<tr>
<td>$!x : \tau$</td>
</tr>
<tr>
<td>Generate/Tell about binding [here]</td>
</tr>
</tbody>
</table>
-binding and equality

$$\exists \tau \overset{=} {=} \tau$$  Type equality constraint

$$\exists \tau$$  Bind an unknown $$\tau$$

$$\exists \tau \overset{=} = \tau$$  Bind $$\tau$$ with current solution.

We can ask “is this solved?”
Context duplications are tree branches!
Building Trees

We build a TCT by traversing the AST

A ‘TCT semantics’: \([T] \tau\)
- Translate \(T\) into a TCT
- Have \(\tau\) become the result type

Retaining all information from the typing rules!
Suppose we want to synthesise a type:
\[ \Gamma^+ \vdash T^+ : \tau^- \]

We use this rule to build this tree:
\[ \Gamma^+, \{ \exists \tau^- \}, \left[ T^+ \right] \tau^+ \]

\( \tau \) acts as a query variable
This typing rule:
\[ x : \tau \in \Gamma \]

\[ \frac{}{\Gamma \vdash x : \tau \text{ Var}} \]

Becomes this TCT rule:
\[
[x^+] \quad \tau^- = \{ ?x^- : \tau^+ \} \quad (\text{Var})
\]
Building Lam

\[
\Gamma f := \Gamma ; x : \tau p^+
\]
\[
\Gamma f \vdash T : \tau r^+
\]
\[
\tau f = \tau p^- \rightarrow \tau r^-
\]

\[
\Gamma \vdash \lambda x. T : \tau f \quad \text{Lam}
\]

\[
[\lambda x. T] \tau f =
\{ \exists \tau p, \exists \tau r, \tau f = \tau p^- \rightarrow \tau r^-, !x : \tau p^+ \}, [T] \tau r^+ \quad \text{(Lam)}
\]
Building App

\[ \Gamma, \Gamma_f, \Gamma_p \vdash T_f : \tau_f \quad \Gamma_p \vdash T_p : \tau_p \]

\[ \tau_p \rightarrow \tau_r = \tau_f \]

\[ \Gamma \vdash T_f \ T_p : \tau_r \quad \text{App} \]

\[ [ T_f \ T_p ] \tau_r = \\
\{ \exists \tau_f, \exists \tau_p, \tau_p \rightarrow \tau_r = \tau_f \} \mid [ T_f ] \tau_f \mid [ T_p ] \tau_p \ (\text{App}) \]
What can I do with them?

We can, of course, implement typecheckers.

We can also instrument those checkers:

- Changes in the tree show the process, including scope
- Those changes preserve the AST structure
  - Mostly – labels are useful extra info
- UI for type level debugging?
What else can I do with them?

We can usually fuse TCTs away:

Assuming we have:
- A `greedySolver` for the TCT’s constraints
- A `treeGenerator`, implementing \([T] \tau\)

We can pick `treeGenerator`’s traversal strategy

And fuse `greedySolver \circ treeGenerator`

Exceptions are interesting! Greed doesn’t always work.
What else can I do with them?

We can find and explain optimisations:

For HM:
(not my trick! – MLton?)

*Generalisation depth* – generalisation constraints from root

- Annotate every $\exists \tau$ with its generalisation depth
- Constraints generalise variables at their depth
- Generalisation is now linear in type size
If you remember one thing...

Telescopic Constraint Trees do in space what conventional checkers do in time

Some extras:

- Information effects track information well
- Constraints are a unit of information
- Information Aware Type Systems
- Telescopic Constraint Trees can be derived
- We can calculate, trade off and optimise from TCTs