Type-based termination analysis with disjunctive invariants

Dimitrios Vytiniotis, MSR Cambridge
with Byron Cook (MSR Cambridge) and Ranjit Jhala (UCSD)

IFIP WG 2.8, Austin TX, March 2011
... or, what am I doing hanging out with these people?

**termination** and liveness of imperative programs, shape analysis and heap space bounds, ranking function synthesis

Program analysis, model checking and verification for systems code, refinement types, **liquid types**, decision procedures

And myself?

Functional programming, **type systems**, type inference, dependent types, semantics and parametricity, Coq, Haskell!
The jungle of termination/totality analysis

"Guarded recursion" (my own term)
- sized types [Hughes et al, Abel]
- modalities for recursion [eg Nakano]

Structural recursion
- Conor McBride
- offered in Coq and Agda
- also Bove & Capretta transformation

Dependent types
- programming with well-founded relations (think “ranking functions”)
- Coq, Agda, DML [Xi]

Size-change principle
- [Jones, Sereni, Bohr]
- a control flow analysis essentially

Terminator
- termination analysis for imperative programs
- “disjunctive invariants” and Ramsey’s theorem
- [Cook, Podelski, Rybalchenko]
A dichotomy?

“Guarded recursion”, structural recursion, dependent types

Terminator and disjunctive invariants, size-change

- 😊 Mostly fully automatic
- 😞 Not programmable
- 😊 No declarative specs
- 😊 Often easy for the tool to synthesize the termination argument

- 😊 Mostly fully manual
- 😊 Programmable
- 😊 Declarative specification
- 😞 Often tedious to come up with a WF relation or convince type checker (i.e. the techniques don’t make proving totality easier, they just make it possible!)

Today I will have a go at combining both worlds

WARNING: very fresh (i.e. airplane-fresh) ideas!
The idea: one new typing rule for totality

\[ T_1 \ldots T_n \text{ well-founded binary relations} \]
\[ dj(a, b) = a <_{T_1} b \lor \ldots \lor a <_{T_n} b \]

\[ \Gamma, (old : T), (g : \{x : T \mid dj(x, old)\} \to U), \]
\[ (x : \{y : T \mid dj(y, old) \lor y = old\}) \vdash e : U \]
\[ \Gamma \vdash fix (\lambda g. \lambda x. e) : T \to U \]
Example

let rec flop (u,v) =
    if v > 0 then flop (u,v-1) else
    if u > 1 then flop (u-1,v) else 1

Terminating, by lexicographic pair order

Γ, (old: T), (g: {x: T | dj(x, old)} → U), (x: {y: T | dj(y, old) ∨ y = old}) ⊢ e: U
Γ ⊢ fix (λg. λx. e): T → U

Consider \( T_1 \ x \ y \equiv \text{fst} x < \text{fst} y \)
Consider \( T_2 \ x \ y \equiv \text{snd} x < \text{snd} y \)  [NOTICE: No restriction on \( \text{fst} \) components!]

Subtyping constraints (obligations) arising from program

\((u, v) = (ou, ov), v > 0 \Rightarrow dj((u, v - 1), (ou, ov))\)  
\((u, v) = (ou, ov), u > 1 \Rightarrow dj((u - 1, v), (ou, ov))\)  
\(dj((u, v), (ou, ov)), v > 0 \Rightarrow dj((u, v - 1), (ou, ov))\)  
\(dj((u, v), (ou, ov)), u > 1 \Rightarrow dj((u - 1, v), (ou, ov))\)
Or ... just call Liquid Types and it will do all that for you!

http://pho.ucsd.edu/liquid/demo/index2.php

... after you have applied a transformation to the original program that I will describe later on
Background

Structural and guarded recursion, dependent types and well-founded relations in Coq

We will skip these. You already know
Background: disjunctive invariants

**Ramsey’s theorem**

*Every infinite complete graph whose edges are colored with finitely many colors contains an infinite monochromatic path.*

**Podelski & Rybalchenko characterization of WF relations**

*Relation $R$ is WF iff there exist WF relations $T_1 \ldots T_n$ such that $R^+ \subseteq T_1 \cup \ldots \cup T_n$*
Background: How Terminator works?

- Transform a program, and assert/infer invariants!

```java
int x = 50;
while (x > 0) do {
    ...
    x = x - 1;
}
```

- Invariant between x and oldx represents any point of R+

```java
bool copied = false;
int oldx;
int x = 50;
while (x > 0) do {
    if copied then
        assert (x <_{T_i} oldx)
    else
        if * then {
            copied=true; oldx=x;
        }
    ...
    x = x - 1;
}
```

- We need non-deterministic choice to allow the “start point” to be anywhere
In a functional setting: a first attempt

- Let’s consider only divergence from recursion
  - Negative recursive types, control ← Not well-thought yet
- The “state” is the arguments of the recursive function
- Hence:

  ```ml
  let rec f x =
  if x==0 then 41 else f (x-1) + 1
  ```

In particular f has to accept x ≤ oldx the first time. But in all subsequent calls it must be x < oldx

```ml
let rec f x =
  if * then
    if x==0 then 41 else f (x-1) + 1
  else
    f' x x
  f' oldx x =
  if x==0 then 41 else f' oldx (x-1) + 1
```

But where is the ASSERT?
In a functional setting: a better attempt

- Just inline the **first** call to \( f' \) to expose subsequent calls:

```ocaml
let rec f x =  
  if x==0 then 41 else f (x-1) + 1
```

```ocaml
let rec f x =   
  if * then   
    if x==0 then 41 else f (x-1) + 1 
  else   
    f' x x if x==0 then 41 else f' x (x-1) + 1

let rec f' oldx x =   
  assert (oldx <= T_i x)  
  if x==0 then 41 else f' oldx (x-1) + 1
```

Starts to look like something a refinement type system could express ... but can we dispense with rewriting?
A special typing rule, to avoid rewriting

\[
\Gamma, (\text{old}: T), (g: \{x: T \mid dj(x, \text{old})\} \rightarrow U), (x: \{y: T \mid dj(y, \text{old}) \lor y = \text{old}\}) \vdash e: U
\]

\[
\Gamma \vdash \text{fix} (\lambda g. \lambda x. e): T \rightarrow U
\]

- A declarative spec of termination with disjunctive invariants

- Given the set \( T_i \) the typing rule can be checked or inferred
  - E.g. inference via Liquid Types [Ranjit]

- It’s a cool thing: programmer needs to come up with simple WF relations (which are also easy to synthesize [Byron])
Bumping up the arguments

```
let rec flop (u,v) =
    if v > 0 then flop (u,v-1) else
    if u > 1 then flop (u-1, big) else 1
```

\[
\Gamma, (old:T), (g:\{x:T \mid dj(x, old)\} \to U), (x:\{y:T \mid dj(y, old) \lor y = old\}) \vdash e:U
\]

\[
\Gamma \vdash fix (\lambda g. \lambda x.e):T \to U
\]

Consider \( T_1(x, y) \equiv fst x < fst y \)
Consider \( T_2(x, y) \equiv snd x < snd y \)
Subtyping constraints (obligations) arising from program

\[
(u, v) = (ou, ov) \land v > 0 \implies dj((u, v - 1), (ou, ov))
\]

\[
(u, v) = (ou, ov) \land u > 1 \implies dj((u - 1, \textbf{big}), (ou, ov))
\]

\[
dj((u, v), (ou, ov)) \land v > 0 \implies dj((u, v - 1), (ou, ov))
\]

\[
dj((u, v), (ou, ov)) \land u > 1 \implies dj((u - 1, \textbf{big}), (ou, ov))
\]
One way to strengthen the rule with invariants

let rec flop (u,v) = 
  if v > 0 then flop (u,v-1) else 
  if u > 1 then flop (u-1, big) else 1

Consider $T_1(x,y) \equiv \text{fst } x < \text{fst } y$
Consider $T_2(x,y) \equiv \text{snd } x < \text{snd } y$ [NOTICE: No restriction on fst!]
Consider $P(x,y) \equiv \text{fst } x \leq \text{fst } y$ [Synthesized or provided]

Subtyping constraints (obligations) arising from program:

$P((u,v),(ou,ov)) \land (u,v) = (ou,ov) \land v > 0 \Rightarrow P((u,v-1),(ou,ov)) \land dj((u,v-1),(ou,ov))$

$P((u,v),(ou,ov)) \land (u,v) = (ou,ov) \land u > 1$
\[ \Rightarrow P((u-1, big),(ou,ov)) \land dj((u-1, big),(ou,ov)) \]

$P((u,v),(ou,ov)) \land dj((u,v),(ou,ov)) \land v > 0 \Rightarrow P((u,v-1),(ou,ov)) \land dj((u,v-1),(ou,ov))$

$P((u,v),(ou,ov)) \land dj((u,v),(ou,ov)) \land u > 1$
\[ \Rightarrow P((u-1, big),(ou,ov)) \land dj((u-1, big),(ou,ov)) \]
Scrap your lexicographic orders? ...

\[ P \text{ reflexive} \]
\[ \Gamma, (\text{old}: T), (g: \{x: T \mid P(x, \text{old}) \land dj(x, \text{old})\} \rightarrow U), \]
\[ (x: \{y: T \mid P(y, \text{old}) \land (dj(y, \text{old}) \lor y = \text{old})\}) \vdash e : U \]
\[ \Gamma \vdash \text{fix} (\lambda g. \lambda x. e) : T \rightarrow U \]

It is arguably very simple to see what \( T_1 \ldots T_n \) are but not as simple to provide a strong enough invariant \( P \)

But the type-system approach may help find this \( P \) interactively from the termination constraints?

... or Liquid Types can infer it for us
What next?

- More examples. Is it easy for the programmer?
- Formal soundness proof
  - Move from trace-based semantics (Terminator) to denotational?
- Integrate in a refinement type system or a dependently typed language
  - Tempted by the Program facilities for extraction of obligations in Coq
  - Is there a constructive proof of (some restriction of) disjunctive WF theorem? If yes, use it to construct the WF ranking relations in Coq
  - Applicable to Agda, Trellys?
  - Liquid types. Demo works for many examples via the transformation
- Negative recursive datatypes, mutual recursion ...
A new typing rule for termination based on disjunctive invariants

New typing rule serves as:

- a declarative specification of that method, or
- the basis for a tool that could potentially increase the programmability of totality checking