TRELLYS and Beyond: Type Systems for Advanced Functional Programming

Aaron Stump

Computer Science
The University of Iowa

Joint work with Tim Sheard, Vilhelm Sjöberg, and Stephanie Weirich.
Supported by NSF grant 0910510.

What is Functional Programming?

Central Ideas:

- Functions as data (higher-order functions, partial applications).
  - `List.map ((+) 10) [ 1 ; 2 ; 3 ]`
- Procedures behave like mathematical functions.
  - monads for making effectful procedures behave.

Culture:

- memory safety (via types)
- compile-time debugging via types
- type inference
- garbage collection
- pattern matching, inductive data
- module systems

The State of FP

Practical ↞— FP —↠ Verifiable

Software Transactional Memory
Lightweight threads
Continuing compiler improvements
...

GADTs, dependent types
Resource typing (HTT)
More powerful type inference
...

Sweet spot, or missing all targets?

This Talk

- Verifiability: TRELLYS
  - practical dependently typed language.
  - quasi-implicit products.
  - briefly: effect system for termination, termination casts.
- Practicality: BLAISE
  - goal: memory-safe functional programming without GC.
  - idea: divide pointers into primary and alias.
  - primary pointers used linearly.
  - alias pointers must have reciprocal backpointer.
  - *early-stage work.*

Design Goals and Design Ideas for TRELLYS

The TRELLYS Project

Goal: a new functional language with dependent types.

- Why Dependent Types Matter™:
  - incremental verification.
  - standard example:

```
[ 10 ; 20 ; 30 ] : vec int 3

append : Forall(A:type)(n1 n2 : nat).
         l1 : vec A n1 -> l2 : vec A n2 -> vec A n1+n2
```

  - small intellectual step from `list A` to `vec A n`.
  - other formal methods: much bigger leap!
- TRELLYS: combine resources to build compiler, libraries, etc.
- Involve broader community (working groups).
- First step: devise core language (PIs + SPJ, CM, BB, WS).

Goals for TRELLYS Core Language

- CBV, mutable state, inductive types, gen. recursion.
- Straightforward syntax-directed type checking.
  - surface language: type inference, automated reasoning.
  - elaboration to core adds annotations.
  - no fancy algorithms in the core.
  - *trustworthy* type checking for core.
- Logically sound fragment under Curry-Howard.
  - general recursion => unsound proof.
  - want a sound notion of proof.
  - without proofs, would need runtime checks.

Technical Goals for TRELLYS Core Language

- General recursive functions, sound proofs.
- Implicit products.
  - specificational ("ghost") data.

```
append : Forall(A:type)(n1 n2 : nat).
         l1 : vec A n1 -> l2 : vec A n2 -> vec A n1+n2
```

  - not computationally relevant.
  - dropped during compilation, and formal reasoning.
- `type:type`
  - most powerful form of polymorphism known.
  - allows type-level computation, data structures:

```
[ int ; bool ; int -> int] : list type
```

- Non-constructive reasoning (AS).
  - already distinguishing terminating/general recursive.
  - so distinguish proofs and programs.
  - motivation: case-split on termination of term.
  - cf. *Logic-Enriched Type Theory*, Zhaohui Luo.

Aside: Non-Constructive Reasoning

- Suppose quantifiers range over values.
- Define `foldr`:

```
(foldr f b [])     = b
(foldr f b (x:xs)) = (f x (foldr f b xs))
```

- Suppose *u* is an assumption of:

```
Forall(a:A)(b:B). (f a b) = (g a b)
```

- Then:

```
(foldr f b l) = (foldr g b l)
```

- Proof by induction on l. Step case:

```
(foldr f b (a:l')) = (f a (foldr f b l'))   // evaluation
                   = (f a (foldr g b l'))   // IH
                   = (g a (foldr g b l'))   // not allowed by u!
                   = (foldr g b (a:l'))
```

- Solution: case split on whether or not `(foldr g b l')` terminates.

Implicit Products

- Want erasure: $|append\ A\ n_1\ n_2\ l_1\ l_2| = append\ l_1\ l_2$.
- For compilation, and for formal reasoning.
- General theoretical approach:
  1. define unannotated system:
     - terms t, types T.
     - type assignment $\Gamma \vdash t : T$, reduction $t \leadsto t'$.
     - type assignment may not be algorithmic.
  2. do metatheory for unannotated system.
  3. define annotated system:
     - annotated terms a, types A.
     - algorithmic typing $\Gamma \Vdash a : A$.
     - erasure $|a| = t$, $|A| = T$.
  4. conclude metatheory for annotated system.

Example: $\mathtt{T}^{\mathtt{vec}}$

- Gödel's System T + equality types, vector types.
- Unannotated terms:

$$\begin{array}{rcl} t & ::= & x \mid (t\ t') \mid \lambda x.t \mid 0 \mid (S\ t) \mid (R_{\mathtt{nat}}\ t\ t'\ t'') \\ & & \mid \mathtt{nil} \mid (\mathtt{cons}\ t\ t') \mid (R_{\mathtt{vec}}\ t\ t'\ t'') \mid \mathtt{join} \end{array}$$

- Reduction relation $t \leadsto t'$ as expected.
- Unannotated types:

$$T\ ::=\ \mathtt{nat} \mid \langle \mathtt{vec}\ T\ t \rangle \mid \Pi x : T.T' \mid \forall x : T.T' \mid t = t'$$

- $\forall x : T.T'$ – implicit product.

Type Assigment: Implicit Products

- A form of intersection type.
- Studied for Implicit Calculus of Constructions [Miquel01].
- No term constructs in unannotated system.

$$\frac{\Gamma, x : T' \vdash t : T \quad x \notin FV(t)}{\Gamma \vdash t : \forall x : T'.T} \qquad \frac{\Gamma \vdash t : \forall x : T'.T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t : [t'/x]T}$$

Type Assigment: Equality Proofs

- Do not rely on an algorithmic conversion relation.
- Use explicit casts to change types of terms.
- Casts are computationally irrelevant.
- So no term construct for unannotated system.

$$\frac{t \downarrow t' \quad \Gamma\ Ok}{\Gamma \vdash \texttt{join} : t = t'} \qquad \frac{\Gamma \vdash t : t_1 = t_2 \quad \Gamma \vdash t' : [t_1/x]T \quad x \notin dom(\Gamma)}{\Gamma \vdash t' : [t_2/x]T}$$

Annotated $\mathtt{T}^{\mathtt{vec}}$

$$a \;::=\; \ldots \mid (a\ a')^- \mid \lambda^- x : A.a \mid (\mathtt{join}\ a\ a') \mid (\mathtt{cast}\ x.A\ a\ a')$$

$$A \;::=\; \mathtt{nat} \mid \langle \mathtt{vec}\ A\ a \rangle \mid \Pi x : A.A' \mid \forall x : A.A' \mid a = a'$$

$$\begin{array}{lcl} |(a\ a')^-| & = & |a| \\ |\lambda^- x : A.a| & = & |a| \\ |(\mathtt{cast}\ x.A\ a\ a')| & = & |a'| \\ |(\mathtt{join}\ a\ a')| & = & \mathtt{join} \end{array}$$

$$\frac{\Gamma \Vdash a : A \quad \Gamma \Vdash a' : A' \quad |a| \downarrow |a'|}{\Gamma \Vdash (\mathtt{join}\ a\ a') : a = a'} \qquad \frac{\Gamma \Vdash a : a_1 = a_2 \quad \Gamma \Vdash a' : [a_1/x]A}{\Gamma \Vdash (\mathtt{cast}\ x.A\ a\ a') : [a_2/x]A}$$

Metatheory

- For unannotated system:
  - Standard results: type preservation, progress.
  - Strong normalization via reducibility argument.
- Lifting results to annotated system straightforward.

Large Eliminations

- Type-level computation.
- Many feel essential to dependent types.

$$T \quad ::= \quad \ldots \mid R\ t\ T\ (\alpha.T')$$

$$\frac{\Gamma \vdash t : R\ 0\ T\ (\alpha.T')}{\Gamma \vdash t : T} \qquad \frac{\Gamma \vdash t : R\ (S\ t')\ T\ (\alpha.T') \quad \Gamma \vdash t' : \mathtt{nat}}{\Gamma \vdash t : [R\ t'\ T\ (\alpha.T')/\alpha]T'}$$

Problem: Inconsistent Contexts

- Well-known issue if can equate types:

$$u : \texttt{nat} = (\Pi x : \texttt{nat}.\texttt{nat}) \;\vdash\; (0\ 0) : \texttt{nat}$$

- Can also type diverging terms.
- Same problem arises with large eliminations.
- Problem is even worse with implicit products:

$$\frac{u : \texttt{nat} = (\Pi x : \texttt{nat}.\texttt{nat}) \;\vdash\; (0\ 0) : \texttt{nat}}{\vdash (0\ 0) : \forall u : \texttt{nat} = (\Pi x : \texttt{nat}.\texttt{nat}).\ \texttt{nat}}$$

- Stuck terms would be typable in empty context!
- Solution: quasi-implicit products.

Quasi-Implicit Products

- Idea: do not completely erase implicit abstraction, application.
- Unannotated system:

$$t \quad ::= \quad \ldots \mid (\lambda.t) \mid (t)$$

$$\frac{\Gamma, x : T' \vdash t : T \quad x \notin FV(t)}{\Gamma \vdash (\lambda.t) : \forall x : T'.T} \quad \frac{\Gamma \vdash t : \forall x : T'.T \quad \Gamma \vdash t' : T'}{\Gamma \vdash (t) : [t'/x]T}$$

- Do not reduce beneath quasi-implicit abstraction.
- Meta-theory (including SN) preserved.
- See "Equality, Quasi-Implicit Products, and Large Eliminations" [pending]

Another Design Sketch: $\mathtt{T}^{\mathtt{eq}\downarrow}$

- "Termination Casts: A Flexible Approach to Termination with General Recursion" [PAR'10].
- Type-and-effect system for termination/possible divergence.
- Include equality types, termination types.
- Termination types reflect the termination effect.
- Terms and types:

$$\begin{array}{lcl} \theta & ::= & \downarrow \mid ? \\ A & ::= & \mathbf{nat} \mid \Pi^{\theta} x : A.A' \mid a = a' \mid \mathbf{Terminates}\ a \end{array}$$

$$\begin{array}{lcl} a & ::= & y \mid a\,a' \mid \lambda y : A\,.\,a \mid 0 \mid \mathbf{S}\,a \\ & \mid & \mathbf{rec_{nat}}\ g(y\ p) : A = a \mid \mathbf{rec}\ g(y : A) : A' = a \mid \mathbf{case}\ x.A\ a\ a'\ a'' \\ & \mid & \mathbf{join}\ a\,a' \mid \mathbf{cast}\ x\,.\,A\,a'\,a \mid \mathbf{terminates}\ a \mid \mathbf{reflect}\ a\,a' \mid \mathbf{inv}\ a\,a' \\ & \mid & \mathbf{contra}\ A\,a \mid \mathbf{abort}\ A \end{array}$$

Termination Casts

- Used to change the effect for a term.
- Proofs of termination first-class.
- External vs. internal verification:
  - Internal: type the function as total.

$$plus \; : \; \Pi x^{\downarrow} : \mathbf{nat}.\Pi y^{\downarrow} : \mathbf{nat}.\mathbf{nat} \; \downarrow$$

  - External: write a proof that the function is total.

$$\begin{array}{lcl} plus & : & \Pi x^{?} : \mathbf{nat}.\Pi y^{?} : \mathbf{nat}.\mathbf{nat} \; ? \\ plus\_tot & : & \Pi x^{\downarrow} : \mathbf{nat}.\Pi y^{\downarrow} : \mathbf{nat}.\mathbf{Terminates} \; (plus \; x \; y) \; \downarrow \end{array}$$

- $\mathtt{T}^{\mathtt{eq}\downarrow}$ supports both.

Selected Typing Rules ($\Gamma \Vdash a : A\,\theta$)

$$\frac{\Gamma \Vdash a : A\,\theta \qquad \Gamma \Vdash a' : \textbf{Terminates}\ a\ \downarrow}{\Gamma \Vdash \textbf{reflect}\, a\, a' : A\,\theta'}\ \text{AT\_REFLECT}$$

$$\frac{\Gamma \Vdash a : A\ \downarrow}{\Gamma \Vdash \textbf{terminates}\, a : \textbf{Terminates}\ a\,\theta}\ \text{AT\_REIFY}$$

$$\frac{\begin{array}{l} p \not\in \textbf{fv}\,|\,a\,| \\ \Gamma' \ = \ \Gamma\,,\, g\ :\ \Pi^{?}x\!:\!\textbf{nat}.A\,,\ y\ :\ \textbf{nat} \\ \Gamma'' \ = \ \Gamma'\,,\, p\ :\ \Pi^{\downarrow}x_1\!:\!\textbf{nat}.\Pi^{\downarrow}u\!:\!y\ =\ \textbf{S}\,x_1.\textbf{Terminates}\ (\,f\,x_1\,) \\ \Gamma'' \Vdash a : [\,y\,/\,x\,]\,A\ \downarrow \end{array}}{\Gamma \Vdash \textbf{rec}_{\textbf{nat}}\ g(y\ p) : A = a : \Pi^{\downarrow}x\!:\!\textbf{nat}.A\,\theta}\ \text{AT\_RECNAT}$$

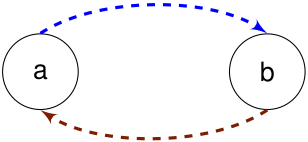
Taming Aliased Structures with Bidirectional Pointers: Blaise

Eliminating Garbage Collection

- GC pros: greater productivity, fewer bugs.
- GC cons: performance hit, unpredictability, complexity.
- Hard to use for real-time systems.
- Resource typing to the rescue?
  - statically track resources.
  - ensure no double deletes, no leaks.
  - alias types, L^3, HTT [Cornell/Harvard PL].
  - Stateful views [Xi et al.]
- Explicitly modeling locations is pretty heavy.
- Alternative: find and enforce safe abstractions for aliased structures.

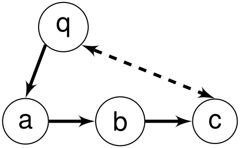
A Safe Abstraction: Bidirectional Pointers

- Idea: divide reference graph into primary/alias pointers.
  - primary pointers should form spanning trees.
  - alias pointers for all other edges.
- Each alias pointer must have a reciprocal backpointer.



- To delete a cell:
  - only delete via a primary pointer.
  - must first disconnect all alias pointers.
  - for disconnected alias pointer: must patch up reciprocal pointer.
- Overhead reasonable: one extra word per alias pointer.
- To enforce this abstraction: symbolic simulation.

Example: FIFO Queues



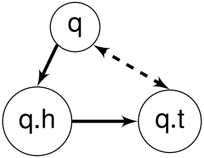

```
type node a
type queue a

inode : all a . Cell (d : a , n : node a) . node a
enode : all a . Cell (d : a , h : queue a) . node a

mk_queue : all a . Cell (h : node a , t : node a) . queue a
empty_queue : all a . Cell () . queue a
```

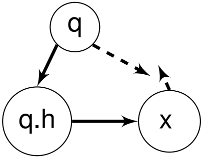
Symbolic Simulation by Example

```
fun insert<a>(consume d : a ,
              update q : queue a).
  (case q of
    empty_queue ->
      let e = new enode <a> in
      e.d = d;
      update q to mk_queue <a>;
      connect e.h q.t;
      q.h = e
  | mk_queue ->
*     let x = disconnect q.t in
      let e = new enode <a> in
      e.d = d;
      connect e.h q.t;
      let y = (take x.d) in
      update x to inode <a>;
      x.d = y;
      x.n = e); 0
```



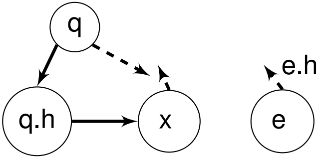
Symbolic Simulation by Example

```
fun insert<a>(consume d : a ,
              update q : queue a).
  (case q of
    empty_queue ->
      let e = new enode <a> in
      e.d = d;
      update q to mk_queue <a>;
      connect e.h q.t;
      q.h = e
  | mk_queue ->
      let x = disconnect q.t in
*     let e = new enode <a> in
      e.d = d;
      connect e.h q.t;
      let y = (take x.d) in
      update x to inode <a>;
      x.d = y;
      x.n = e); 0
```



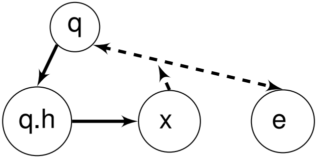
Symbolic Simulation by Example

```
fun insert<a>(consume d : a ,
              update q : queue a).
  (case q of
    empty_queue ->
      let e = new enode <a> in
      e.d = d;
      update q to mk_queue <a>;
      connect e.h q.t;
      q.h = e
  | mk_queue ->
      let x = disconnect q.t in
      let e = new enode <a> in
      e.d = d;
*     connect e.h q.t;
      let y = (take x.d) in
      update x to inode <a>;
      x.d = y;
      x.n = e); 0
```



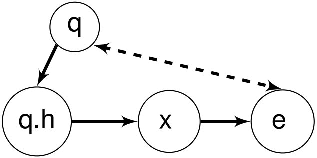
Symbolic Simulation by Example

```
fun insert<a>(consume d : a ,
              update q : queue a).
  (case q of
    empty_queue ->
      let e = new enode <a> in
      e.d = d;
      update q to mk_queue <a>;
      connect e.h q.t;
      q.h = e
  | mk_queue ->
      let x = disconnect q.t in
      let e = new enode <a> in
      e.d = d;
      connect e.h q.t;
*     let y = (take x.d) in
      update x to inode <a>;
      x.d = y;
      x.n = e); 0
```



Symbolic Simulation by Example

```
fun insert<a>(consume d : a ,
              update q : queue a).
  (case q of
    empty_queue ->
      let e = new enode <a> in
      e.d = d;
      update q to mk_queue <a>;
      connect e.h q.t;
      q.h = e
  | mk_queue ->
      let x = disconnect q.t in
      let e = new enode <a> in
      e.d = d;
      connect e.h q.t;
      let y = (take x.d) in
      update x to inode <a>;
      x.d = y;
      x.n = e); 0
*
```



Conclusion

- FP and Verifiability: TRELLYS:
  - combine dependent types, general recursion.
  - quasi-implicit products.
  - termination casts.
  - *next step:* putting it all together, with `type:type`.
- FP and Practicality: BLAISE:
  - programming discipline: primary/alias pointers, reciprocal alias pointers.
  - enforce by symbolic simulation.
  - *next step*: define symbolic simulation, implement.
- For more info:
  - "Equality, Quasi-Implicit Products, and Large Eliminations"
  - "Termination Casts: A Flexible Approach to Termination with General Recursion"
  - See also QA9 (blog).

Thanks for having me!