The Essence of Generalized Algebraic Data Types

Filip Sieczkowski [1] **Sergei Stepanenko** [2] Jonathan Sterling [3]
Lars Birkedal [2]

[1]Heriot-Watt University

[2]Aarhus University

[3]University of Cambridge

November 30, 2023

Table of Contents

1. Motivation
    - Informal description
    - Languages with GADTs
    - Examples
    - State-of-the-Art
2. Language
    - Intuition
    - Definitions
    - Example programs
3. Semantics
    - First attempt
    - Second attempt
4. Conclusion
    - Contributions and future

Type-indexed vectors

```
data Zero :: *
data Succ :: * -> *

data Vec :: * -> * -> * where
  Nil :: forall a. Vec a Zero
  Cons :: forall a n. a -> Vec a n -> Vec a (Succ n)
```

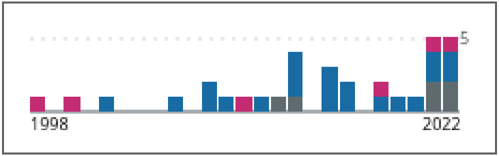
Fixed-length lists without dependent types.

Well-typed DSLs

```
data Expr :: * -> * where
  LiftExpr :: forall a. a -> Expr a
  LamExpr :: forall a b. (a -> Expr b) -> Expr (a -> b)
  AppExpr :: forall a b. Expr (a -> b)
                         -> Expr a -> Expr b
  FixExpr :: forall a. Expr (a -> a) -> Expr a
  ...
```

Can be used to embed DSLs, and fallback onto Haskell for typechecking.
E.g., Pugs, Darcs.

Existing works on GADTs



31 relevant results on dblp:

- 8: categorical semantics
- 23: syntax, type-inference, implementation, usage

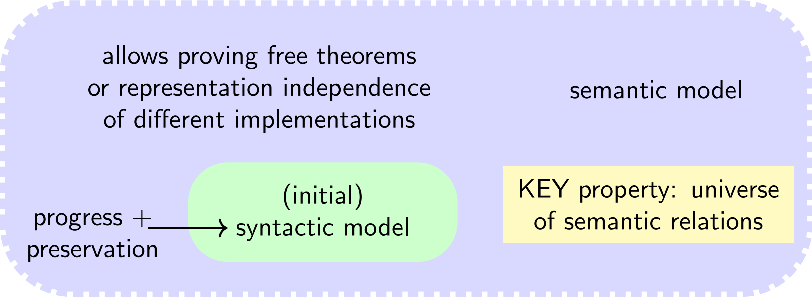
- Our work can be used by compiler developers, language designers, or for the verification of data structures.
- Moreover, we target feature-rich languages.

This work

- Calculus for GADTs.
- Semantic model for this calculus.

This work

- Calculus for GADTs.
- Semantic model for this calculus.



allows proving free theorems
or representation independence
of different implementations

semantic model

progress + preservation → (initial) syntactic model

KEY property: universe of semantic relations

Short description

Our calculus:
System $F\omega$
- $+$ recursive types
- $+$ type equalities
- $+$ optional additional type constructors

Can be perceived as an IR used by a compiler.

Syntax

$$
\begin{array}{llll}
\text{kinds} & \kappa & ::= & \mathsf{T} \mid \kappa \Rightarrow \kappa \\
\text{constructors} & \mathsf{c} & ::= & \forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \mid \to \mid \times \mid + \mid \mathsf{unit} \mid \mathsf{void} \\
\text{constraints} & \chi & ::= & \sigma \equiv_\kappa \tau \\
\text{types} & \tau, \sigma & ::= & \alpha \mid \lambda\alpha :: \kappa.\ \tau \mid \sigma\ \tau \mid \mathsf{c} \mid \chi \to \tau \mid \chi \times \tau
\end{array}
$$

Syntax

kinds $\kappa ::= \mathsf{T} \mid \kappa \Rightarrow \kappa$

constructors $c ::= \forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \mid {\to} \mid \times \mid + \mid \mathsf{unit} \mid \mathsf{void}$

constraints $\chi ::= \sigma \equiv_\kappa \tau$

types $\tau, \sigma ::= \alpha \mid \lambda\alpha :: \kappa.\ \tau \mid \sigma\ \tau \mid c \mid \chi \to \tau \mid \chi \times \tau$

values $v ::= x \mid \langle\rangle \mid \lambda x.\ e \mid \langle v_1, v_2\rangle \mid \mathsf{inj}_1\ v \mid \mathsf{inj}_2\ v \mid \Lambda.\ e$
$\mid \mathsf{pack}\ v \mid \mathsf{roll}\ v \mid \lambda\bullet.\ e \mid \langle\bullet, v\rangle$

expressions $e ::= \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid v_1\ v_2 \mid \mathsf{proj}_1\ v \mid \mathsf{proj}_2\ v$
$\mid \mathsf{case}\ v\ [x.\ e_1 \mid y.\ e_2] \mid v \mid \mathsf{abort}\ \bullet \mid v\ \bullet$
$\mid \mathsf{let}\ (\bullet, x) = v\ \mathsf{in}\ e$

eval. contexts $E ::= \square \mid \mathsf{let}\ x = E\ \mathsf{in}\ e$

- Type constructors are built-in functions on types.
- Constraint types are 'assert's and 'assume's for type equalities.

Dynamic semantics

Constraints don't have any interesting operational semantics, and their introduction and elimination forms just 'guide typechecking'.

$$(\lambda \bullet .\ e)\ \bullet \ \mapsto\ e$$

Dynamic semantics

Constraints don't have any interesting operational semantics, and their introduction and elimination forms just 'guide typechecking'.

$$(\lambda \bullet.\ e)\ \bullet \mapsto\ e$$

$$\mathsf{let}\,(\bullet, x) = \langle \bullet, v\rangle\,\mathsf{in}\,e \mapsto\ e[v/x]$$

Dynamic semantics

Constraints don't have any interesting operational semantics, and their introduction and elimination forms just 'guide typechecking'.

$$(\lambda \bullet.\ e)\ \bullet\ \mapsto\ e$$

$$\mathsf{let}\,(\bullet, x) = \langle \bullet, v\rangle\ \mathsf{in}\ e\ \mapsto\ e[v/x]$$

Informally, we can consider constraints as singleton types.

Discriminability

For impossible case elimination it is enough to look at the head symbols.

$$\frac{c_1 \neq c_2 \qquad (\Delta \vdash c_i\ \overline{\tau}_i :: \kappa)_{i \in \{1,2\}}}{\Delta \Vdash c_1\ \overline{\tau}_1 \#_{\kappa}\ c_2\ \overline{\tau}_2}$$

Discriminability

For impossible case elimination it is enough to look at the head symbols.

$$\frac{c_1 \neq c_2 \qquad (\Delta \vdash c_i\ \overline{\tau}_i :: \kappa)_{i \in \{1,2\}}}{\Delta \Vdash c_1\ \overline{\tau}_1 \#_\kappa\ c_2\ \overline{\tau}_2}$$

$$\frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta \vdash \tau_2 :: \mathsf{T} \qquad \Delta \vdash \sigma_1 :: \mathsf{T} \qquad \Delta \vdash \sigma_2 :: \mathsf{T}}{\Delta \Vdash \tau_1 + \sigma_1 \#_{\mathsf{T}}\ \tau_2 \times \sigma_2}$$

Provability

The main crux of the system — injectivity. This rule makes it possible to use GADTs (e.g., in case of well-typed terms) to derive non-trivial equalities.

$$\frac{c :: (\kappa_i \Rightarrow)_i\, \kappa \qquad \Delta \mid \Phi \Vdash c\ (\sigma_i)_i \equiv_\kappa c\ (\tau_i)_i}{(\Delta \mid \Phi \Vdash \sigma_i \equiv_{\kappa_i} \tau_i)_i}$$

Provability

The main crux of the system — injectivity. This rule makes it possible to use GADTs (e.g., in case of well-typed terms) to derive non-trivial equalities.

$$\frac{c :: (\kappa_i \Rightarrow)_i\, \kappa \qquad \Delta \mid \Phi \Vdash c\ (\sigma_i)_i \equiv_\kappa c\ (\tau_i)_i}{(\Delta \mid \Phi \Vdash \sigma_i \equiv_{\kappa_i} \tau_i)_i}$$

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \times \tau_1 \equiv_{\mathsf{T}} \sigma_2 \times \tau_2}{\Delta \mid \Phi \Vdash \tau_1 \equiv_{\mathsf{T}} \tau_2}$$

Static semantics

We handle constraint passing manually to simplify semantic model.
This is handled by the typechecker in real calculi with GADTs.

$$\frac{\Delta \vdash \chi \text{ constr} \qquad \Delta \mid \Phi, \chi \mid \Gamma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \lambda \bullet.\ e : \chi \to \tau} \qquad \frac{\Delta \mid \Phi \Vdash \chi \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \langle \bullet, v \rangle : \chi \times \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \equiv_\kappa \sigma_2 \qquad \Delta \Vdash \sigma_1 \#_\kappa \sigma_2 \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \mid \Phi \mid \Gamma \vdash \text{abort} \bullet : \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \tau_1 \equiv_{\mathsf{T}} \tau_2 \qquad \Delta \mid \Phi \mid \Gamma \vdash e : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash e : \tau_2}$$

Type-indexed vectors

Some classical examples of dependent types can be expressed with just GADTs, if we use them as 'tags'.

$$
\begin{aligned}
&\mathsf{natvec} :: \mathsf{T} \Rightarrow \mathsf{T}\\
&\mathsf{natvec} \triangleq\\
&\quad \mu\varphi :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha :: \mathsf{T}.\\
&\quad ((\alpha \equiv_{\mathsf{T}} \mathsf{void}) \times \mathsf{unit})\\
&\quad +(\mathbb{N} \times \exists\beta :: \mathsf{T}.\ (\alpha \equiv_{\mathsf{T}} (\beta + \mathsf{unit})) \times (\varphi\ \beta))
\end{aligned}
$$

natvec is either unit (and has void as its index)
or not unit (and the tail has a smaller index).

$$
\begin{aligned}
&\mathsf{nenatvec} :: \mathsf{T}\\
&\mathsf{nenatvec} \triangleq \exists\alpha :: \mathsf{T}.\ \mathsf{natvec}\ (\alpha + \mathsf{unit})
\end{aligned}
$$

Type-indexed vectors

The head function is now total! (We can eliminate the impossible case.)

$$
\begin{array}{l}
\mathsf{vhead} : \mathsf{nenatvec} \to \mathbb{N} \\
\mathsf{vhead}\, xs \triangleq \\
\quad \mathsf{let}\,(*, ys) = xs\,\mathsf{in} \\
\quad \mathsf{case}\ \mathsf{unroll}\, ys \\
\quad \mid \mathsf{inj}_1\,(\bullet, w).\ \mathsf{abort}\,\bullet \\
\quad \mid \mathsf{inj}_2\,\langle y, \_\rangle.\ y
\end{array}
$$

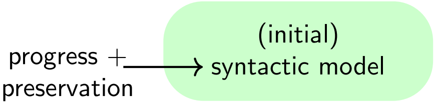
Enough about the syntax of our calculus! Can we actually prove something about it?

Naïve approach

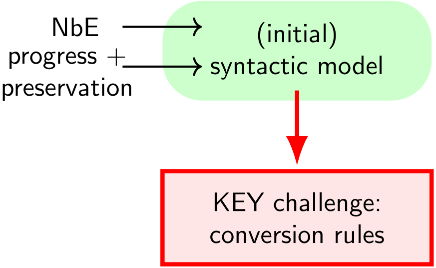
- Types are interpreted as sets of values. Constraints are interpreted as equalities of these sets.
- We can't validate injectivity rules, e.g., consider this instance:

$$\frac{\Delta \mid \Phi \Vdash \mathsf{void} \times \tau_1 \equiv_{\mathsf{T}} \mathsf{void} \times \tau_2}{\Delta \mid \Phi \Vdash \tau_1 \equiv_{\mathsf{T}} \tau_2}$$

- If $\emptyset \times A = \emptyset \times B$, then it isn't necessarily true that $A = B$.



progress + preservation $\longrightarrow$ (initial) syntactic model

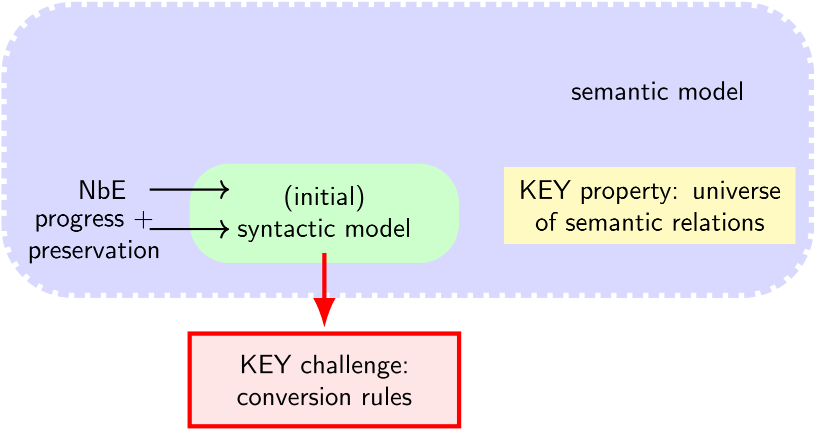


NbE

progress + preservation

(initial) syntactic model

KEY challenge: conversion rules



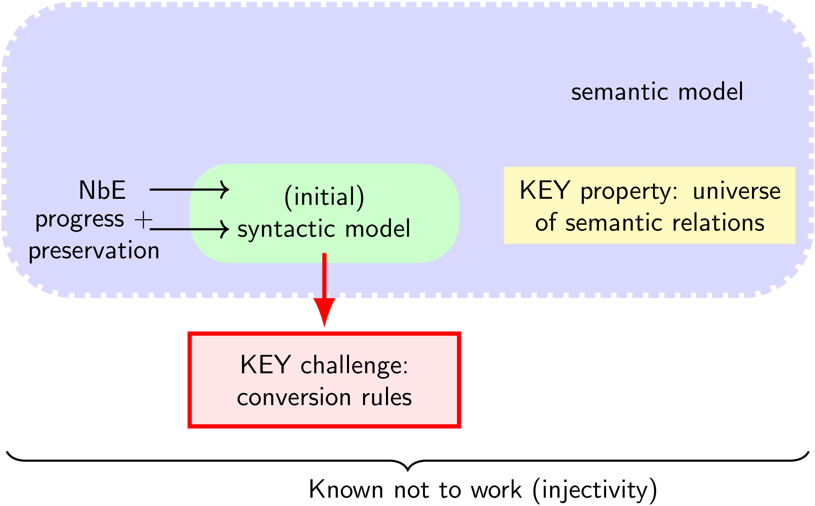
semantic model

NbE ⟶ (initial) syntactic model

progress + preservation ⟶ (initial) syntactic model

KEY property: universe of semantic relations

KEY challenge: conversion rules



semantic model

NbE

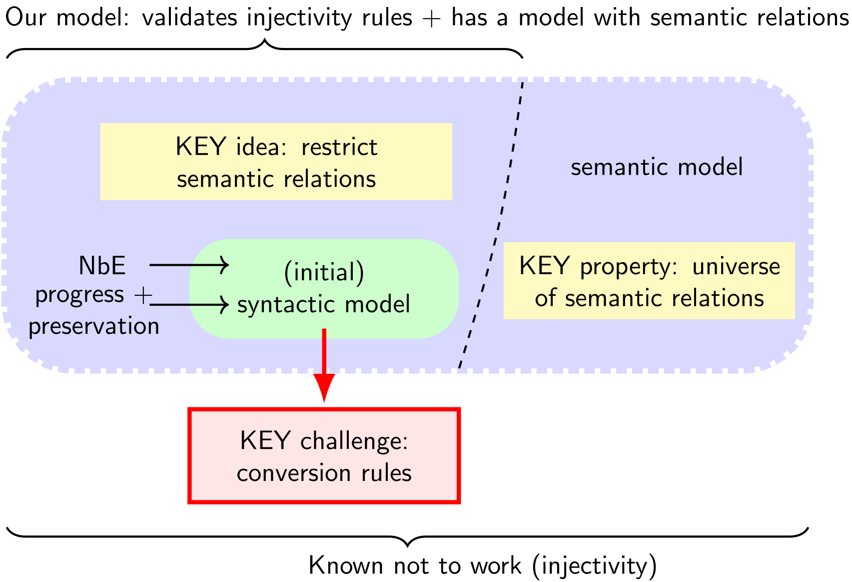
progress + preservation

(initial) syntactic model

KEY property: universe of semantic relations

KEY challenge: conversion rules

Known not to work (injectivity)



Our model: validates injectivity rules + has a model with semantic relations

KEY idea: restrict semantic relations

semantic model

NbE
progress + preservation

(initial) syntactic model

KEY property: universe of semantic relations

KEY challenge: conversion rules

Known not to work (injectivity)

High-level explanation

Idea: two stages.

- Before we compute the actual semantics, we need to do some equalities house-keeping.
- The first stage helps to reason about equalities.
- The second stage is for sets of values.

Normal forms

- We need an inductively defined universe of ‘codes’ for types.
- We can use NbE for types, and their normal forms as codes.

First stage

- *Normalization by evaluation* for types.
- Syntax of normal and neutral forms for types.
- Normalization is performed by composition of reify and eval.

$$\begin{aligned} [\![\mathsf{T}]\!] &\triangleq \mathrm{Neu}_{\mathsf{T}} \\ [\![\kappa_a \Rightarrow \kappa_r]\!] &\triangleq [\![\kappa_a]\!] \Rightarrow [\![\kappa_r]\!] \\ [\![\Delta]\!] &\triangleq \prod_{\alpha::\kappa \in \Delta} [\![\kappa]\!] \end{aligned}$$

$$\begin{aligned} \mathrm{reify} &: [\![\kappa]\!] \Rightarrow \mathrm{Nf}_{\kappa} \\ \mathrm{reflect} &: \mathrm{Neu}_{\kappa} \Rightarrow [\![\kappa]\!] \\ \mathrm{eval} &: \mathrm{Ty}^{\Delta}_{\kappa} \to ([\![\Delta]\!] \Rightarrow [\![\kappa]\!]) \end{aligned}$$

Setup for the second stage

We used step-indexed logic for this version of the calculus.
Language features might require additional gadgets.

$$\tau ::= T \mid \mathit{Val} \mid \mathit{Expr} \mid \mathrm{Prop} \mid 1 \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau$$

$$\begin{aligned} t, P ::= {} & x \mid v \mid e \mid F(t_1, \ldots, t_n) \mid \\ & () \mid (t, t) \mid \pi_i\ t \mid \lambda x : \tau.\, t \mid t(t) \mid \\ & \mathrm{inl}\ t \mid \mathrm{inr}\ t \mid \mathrm{case}(t, x.t, y.t) \mid \\ & \mathrm{False} \mid \mathrm{True} \mid t =_\tau t \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \\ & \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \triangleright P \mid \mu\, x : \tau.\, t \end{aligned}$$

$$\frac{\Gamma, x : \tau \vdash t : \tau \qquad x \text{ is guarded in } t}{\Gamma \vdash \mu\, x : \tau.\, t : \tau}$$

Second stage

- We can interpret normal forms now, instead of arbitrary types.
- Syntactic equality of normal forms for constraints.

$$(\Delta \vdash \tau \equiv_\kappa \sigma \ \mathsf{constr})\,\mathrm{true} \triangleq$$
$$\forall(\Delta' : \mathrm{Ctx})(\eta : [\![\Delta]\!]^{\Delta'}).\mathrm{reify}(\mathrm{eval}(\tau)(\eta)) = \mathrm{reify}(\mathrm{eval}(\sigma)(\eta))$$

$$\mathcal{R}(\chi \times \nu)(\mathsf{v}) \triangleq \exists \mathsf{v}'.\mathsf{v} = \langle \bullet, \mathsf{v}' \rangle \wedge \chi\,\mathrm{true} \wedge \mathcal{R}(\nu)(\mathsf{v}')$$

But what happens when we want to go under binders in types
(at this point, it's either forall or recursive types)?

$$\mathcal{R}(\forall\alpha :: \kappa.\ \tau)(v) \triangleq \exists e.\ v = \Lambda.\ e \wedge \forall\mu \in ?.\mathrm{wp}(\mathcal{R}(\tau + \text{something about } \mu \text{ and } \alpha))(e)$$

But what happens when we want to go under binders in types (at this point, it's either forall or recursive types)?

$$\mathcal{R}(\forall\alpha :: \kappa.\ \tau)(v) \triangleq \exists e.\ v = \Lambda.\ e \wedge \forall\mu \in ?.\mathrm{wp}(\mathcal{R}(\tau + \text{something about } \mu \text{ and } \alpha))(e)$$

What if we want to validate an equality in τ that involves α?

If we try to use predicates, we end up with a function from an interpretation of $\mathbb{N}$ to some arbitrary interpretation of α and a syntactic constraint.

$$\mathcal{R}(\forall\alpha :: \mathsf{T}.\ \forall\beta :: \mathsf{T}.\ (\alpha \times \beta \equiv_{\mathsf{T}} \mathbb{N} \times \beta) \to \mathbb{N} \to \alpha)$$

Second stage*

- We cannot directly use predicates for ∀.
- Guarded recursion not only in the case of recursive types, but also in the case of ∀ (syntactic substitution strikes back).
- We can interpret normal forms now, instead of arbitrary types.
- Syntactic equality of normal forms for constraints.

$$\mathcal{R}(\forall\alpha :: \kappa.\ \tau)(v) \triangleq \exists e.\ v = \Lambda.\ e \wedge \forall\mu \in [\![\kappa]\!](\cdot).\triangleright\mathrm{wp}(\mathcal{R}(\mathrm{eval}(\tau)([\alpha \mapsto \mu])))(e)$$

$$\mathcal{R}(\chi \times \nu)(v) \triangleq \exists v'.v = \langle \bullet, v' \rangle \wedge \chi\,\mathrm{true} \wedge \mathcal{R}(\nu)(v')$$

Dealing with equalities

$$(\Delta \vdash \tau \equiv_\kappa \sigma \ \mathsf{constr}) \, \mathrm{true} \triangleq$$
$$\forall(\Delta' : \mathrm{Ctx})(\eta : [\![\Delta]\!]^{\Delta'}).\mathrm{reify}(\mathrm{eval}(\tau)(\eta)) = \mathrm{reify}(\mathrm{eval}(\sigma)(\eta))$$

- To validate the conversion rule we need reification to be injective.
- That's what *good* stands for.

Lemma (Injectivity of reify)

For any types τ_1, τ_2 *of kind* κ*, well-formed in* Δ
and any good environment $\eta : [\![\Delta]\!]^{\Delta'}$,
if $\mathrm{reify}(\mathrm{eval}(\tau_1)(\eta)) = \mathrm{reify}(\mathrm{eval}(\tau_2)(\eta))$*, then* $\mathrm{eval}(\tau_1)(\eta) = \mathrm{eval}(\tau_2)(\eta)$.

Second stage**

- We need to maintain a *good* environment.
- We cannot use purely semantic predicates in ∀.
- Guarded recursion not only in case of recursive types, but also in ∀ (syntactic substitution strikes back).
- We can interpret normal forms now, instead of arbitrary types.
- Syntactic equality of normal forms for constraints.

$$\mathcal{R}(\forall\alpha :: \kappa.\ \tau)(v) \triangleq \exists e.\ v = \Lambda.\ e \wedge \forall\mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \rightarrow \triangleright\mathrm{wp}(\mathcal{R}(\mathrm{eval}(\tau)([\alpha \mapsto \mu])))(e)$$

$$\mathcal{R}(\chi \times \nu)(v) \triangleq \exists v'.v = \langle\bullet, v'\rangle \wedge \chi\,\mathrm{true} \wedge \mathcal{R}(\nu)(v')$$

Logical relation

We avoided a few problems so far:

- If the interpretation of equalities is too *semantical*, we cannot validate injectivity rules.
- If we use equalities of normal forms to interpret equalities, but do not use *good* environments, we cannot validate the conversion rules.

After that, we can actually validate all the rules, given a *good* environment for types, and valid contexts for constraints and term variables.

You might have a question: it's good, but is it usable for anything apart from proving safety in a semantical way?

You might have a question: it's good, but is it usable for anything apart from proving safety in a semantical way?

No!

You might have a question: it's good, but is it usable for anything apart from proving safety in a semantical way?

No!

But we can cook up something.

We can extend the syntax with arbitrary stuff at the base kind (remember, that reify and reflect are ‘inert’ for the base kind).

$$\frac{\varphi : X}{\varphi : \mathrm{Neu}^{\mathsf{\Delta}}_{\mathsf{T}}}$$

If X is instantiated with predicates (or relations), we can verify syntactically not well-typed programs (or prove interesting binary properties).

To evaluate the model we prove representation independence of type-indexed vectors and lists.

Moreover, we don't break anything in the process (e.g., we still can prove free theorems).

Limitations

- The model is not robust enough to ensure termination of the sub-calculus with restricted injectivity and no recursive types.
- Relational reasoning is limited to types at the base kind (well, in any case, programmers don't use types at higher kinds).

Contributions:

- Calculus for studies of GADTs.
- Novel approach to study semantics of feature-rich languages with syntactic constraints for types.
- Semantical models of a language that allows us to express GADTs:
  - Unary model that validates potential extensions for languages with GADTs.
  - Binary model that allows reasoning about representation independence (and (sic!) doesn't break anything from System F).

and future:

- Extensions (general effects).
- Relational interpretation of $\forall$ quantified at higher kinds. (Less restrictive interpretation of $\forall$?)
- The end goal is to provide a setup that can be used for designing a language that supports GADTs.

Placeholder before backup slides

[redacted]

Logical relation

$$\begin{aligned}
[\![\Phi]\!]_\eta\,\mathrm{true} &\triangleq \forall \varphi \in \Phi.\ [\![\varphi]\!]_\eta\,\mathrm{true} \\
[\![\Gamma]\!]_\eta &\triangleq \{\gamma \in \mathrm{dom}(\Gamma) \to \mathrm{Val} \mid \forall x \in \mathrm{dom}(\Gamma).\ \mathcal{R}(\mathrm{eval}(\Gamma(x))(\eta))(\gamma(x))\} \\
\Delta \mid \Phi \mid \Gamma \models e : \tau &\triangleq \forall \eta \in [\![\Delta]\!](\cdot).\ \mathrm{good}(\eta) \to [\![\Phi]\!]_\eta\,\mathrm{true} \to \forall \gamma \in [\![\Gamma]\!]_\eta \to \mathrm{wp}(\mathcal{R}(\mathrm{eval}(\tau)(\eta)))(e)
\end{aligned}$$

Injectivity and Cantor's paradox

Injectivity of some constructors implies false. It's a known fact, but can come up as a surprise.

For any injective constructor $c :: (\mathsf{T} \Rightarrow \mathsf{T}) \Rightarrow \mathsf{T}$ and type $\alpha :: \mathsf{T}$ it is possible to derive a value of type void in System $\mathsf{F}_{\omega}^{=i}$.

[redacted]

Non-termination

For any injective constructor $c :: (\mathsf{T} \Rightarrow \mathsf{T}) \Rightarrow \mathsf{T}$ and type $\alpha :: \mathsf{T}$ it is possible to derive a value of type void in System $\mathsf{F}_{\omega}^{=i}$.

- $\tau_c^{\mathrm{loop}} \triangleq \exists\beta :: \mathsf{T} \Rightarrow \mathsf{T}.\ (c\ \beta \equiv_{\mathsf{T}} \alpha) \times (\beta\ \alpha \rightarrow \mathsf{void})$
- $v^{\mathrm{loop}} \triangleq \lambda x.\ \mathsf{let}\,(*, (\bullet, y)) = x\,\mathsf{in}\,y\ (\mathsf{pack}\,\langle \bullet, y\rangle)$
- $\vdash v^{\mathrm{loop}} : \tau_c^{\mathrm{loop}}[(c\ (\lambda\alpha :: \mathsf{T}.\ \tau_c^{\mathrm{loop}}))/\alpha] \rightarrow \mathsf{void},$
- $\vdash v^{\mathrm{loop}}\ (\mathsf{pack}\,\langle \bullet, v^{\mathrm{loop}}\rangle) : \mathsf{void}$

[redacted]

Syntactic type-safety via NbE

Lemma (Consistency)

A discriminable constraint is not provable in an empty context: in other words, $\emptyset \mid \emptyset \Vdash \tau_1 \equiv_\kappa \tau_2$ *and* $\emptyset \Vdash \tau_1 \#_\kappa \tau_2$ *are contradictory.*

- Consequence of the injectivity of reify.
- Allows to discharge impossible cases.

Lemma (Canonical form for arrows)

If v *is a closed value of type* τ *and* τ *is provably equal to some arrow type in an empty context, then* v *is a lambda-abstraction with a well-typed body.*

$$(\emptyset \mid \emptyset \Vdash \tau \equiv_{\mathsf{T}} (\tau_1 \to \tau_2)) \wedge (\emptyset \mid \emptyset \mid \Gamma \vdash v : \tau)$$
$$\implies (\exists x e.\ v = \lambda x.\ e \wedge \emptyset \mid \emptyset \mid \Gamma, x : \tau_1 \vdash e : \tau_2)$$

Orthogonal extensions

References and concurrency.

$$\begin{array}{lcll}
\text{constructors} & c & ::= & \cdots \mid \mathsf{ref} \\
\text{references} & \ell & ::= & \mathbb{N} \\
\text{values} & v & ::= & \cdots \mid \ell \\
\text{expressions} & e & ::= & \cdots \mid \mathsf{fork}\ e \mid \mathsf{alloc}\ v \mid v := v \mid !\ v
\end{array}$$

The first stage stays the same, and the rest depends only on the logic used for defining $\mathcal{R}$.

The only requirements are that new effects should be expressed by type constructors, and that the ambient logic can express them.

[redacted]

Type-safe red-black trees

```
data Red
data Black
data Tree a where
  Tree :: Node Black n a -> Tree a

data Node t n a where
  Nil :: Node Black Zero a
  BlackNode :: NodeH t0 t1 n a -> Node Black (Succ n) a
  RedNode :: NodeH Black Black n a -> Node Red n a
data NodeH l r n a = NodeH (Node l n a) a (Node r n a)
```

Stronger type invariants.

Well-typed lambda terms

$$
\begin{aligned}
&\mathsf{Tm} :: \mathsf{T} \Rightarrow \mathsf{T}\\
&\mathsf{Tm} \triangleq\\
&\quad \mu\varphi :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha :: \mathsf{T}.\\
&\quad \alpha + (\exists\beta,\gamma :: \mathsf{T}.\ (\alpha \equiv_{\mathsf{T}} (\beta \to \gamma)) \times (\beta \to \varphi\ \gamma))\\
&\qquad + (\exists\beta :: \mathsf{T}.\ \varphi\ (\beta \to \alpha) \times \varphi\ \beta)
\end{aligned}
$$

[redacted]

Well-typed lambda terms

$$
\begin{array}{l}
\mathsf{eval} : \forall\alpha :: \mathsf{T}.\ \mathsf{Tm}\ \alpha \to \alpha \\
\mathsf{eval} \triangleq \\
\quad \mathsf{fix}\, \lambda f.\ \Lambda.\ \lambda x. \\
\quad \mathsf{case}\ \mathsf{unroll}\, x \\
\quad \mid \mathsf{inj}_1\, y.\ y \\
\quad \mid \mathsf{inj}_2\, y.\ \mathsf{case}\ y \\
\qquad \mid \mathsf{inj}_1\, (*, (*, (\bullet, g))).\ \lambda z.\ f * (g\ z) \\
\qquad \mid \mathsf{inj}_2\, (*, \langle g, x \rangle).\ (f * g)\ (f * x)
\end{array}
$$

[redacted]

Logical relation for denotations

For any two bigger related contexts and arguments in this extended contexts, results are related after extension.

$$\begin{aligned}\eta \mid \nu_1 \approx_{\mathsf{T}} \nu_2 &\triangleq [\![\nu_1]\!]_\eta = \nu_2\\ \eta \mid \varphi_1 \approx_{\kappa_a \Rightarrow \kappa_r} \varphi_2 &\triangleq \forall \Delta_1', \Delta_2', (\delta_1 : \mathsf{hom}_{\mathcal{K}}(\Delta_1', \Delta_1), \delta_2 : \mathsf{hom}_{\mathcal{K}}(\Delta_2', \Delta_2)), (\eta' : [\![\Delta_1']\!]^{\Delta_2'}), \mu_1, \mu_2.\\ &\qquad (\delta_2^* \eta = \lambda \mathsf{x}.\ \eta'(\delta_1(\mathsf{x}))) \to (\eta' \mid \mu_1 \approx_{\kappa_a} \mu_2) \to (\eta' \mid \varphi_1(\delta_1, \mu_1) \approx_{\kappa_r} \varphi_2(\delta_2, \mu_2))\end{aligned}$$

Lemma

If $\eta \mid \mu_1 \approx \mu_2$, *then* $[\![\mathrm{reify}(\mu_1)]\!]_\eta = \mu_2$.
If $\eta \mid \eta_1 \approx \eta_2$, *then* $\eta \mid [\![\tau]\!]_{\eta_1} \approx [\![\tau]\!]_{\eta_2}$.

$$\imath \mid \nu \approx_\kappa \nu$$