

HASCASL: Integrated Specification and Development of Functional Programs

Lutz Schröder and Till Mossakowski



Introduction

Introduction

- HASCASL is a conservative extension of CASL :

Introduction

- **HASCASL** is a conservative extension of **CASL**:
 - Partial λ -calculus
 - Type-class oriented shallow polymorphism

Introduction

- HASCASL is a conservative extension of CASL:
 - Partial λ -calculus
 - Type-class oriented shallow polymorphism
- Aim: specification and functional programming within a single language

Introduction

- **HASCASL** is a conservative extension of **CASL**:
 - Partial λ -calculus
 - Type-class oriented shallow polymorphism
- **Aim**: specification and functional programming within a single language
- **This works via a bootstrap method**:
 - Internal Logic
 - HOLCF-style fixed-point recursion

The Partial λ -Calculus

The Partial λ -Calculus

- Terms need not denote

The Partial λ -Calculus

- Terms need not denote
- **Partial** function types $s_1 \dots s_n \rightarrow ?t$

The Partial λ -Calculus

- Terms need not denote
- **Partial** function types $s_1 \dots s_n \rightarrow ?t$
- λ -abstraction produces **partial** functions

The Partial λ -Calculus

- Terms need not denote
- **Partial** function types $s_1 \dots s_n \rightarrow ?t$
- λ -abstraction produces **partial** functions
- correspondingly geared deduction

The Partial λ -Calculus

- Terms need not denote
- **Partial** function types $s_1 \dots s_n \rightarrow ?t$
- λ -abstraction produces **partial** functions
- correspondingly geared deduction
- **Predicates** are partial functions into unit,

The Partial λ -Calculus

- Terms need not denote
- **Partial** function types $s_1 \dots s_n \rightarrow ?t$
- λ -abstraction produces **partial** functions
- correspondingly geared deduction
- **Predicates** are partial functions into unit, but. . .

The Partial λ -Calculus

- Terms need not denote
- **Partial** function types $s_1 \dots s_n \rightarrow ?t$
- λ -abstraction produces **partial** functions
- correspondingly geared deduction
- **Predicates** are partial functions into unit, but. . .
- **logic** within λ -abstractions initially limited to truth, conjunction

Semantics

Semantics

- Models are **syntactical λ -algebras** (e.g. Breazu-Tannen/Meyer 1985):
 - Interpret **all terms** as partial functions
 - Substitution = composition, variables = projections
 - **Require** compatibility with deduction

Semantics

- Models are **syntactical λ -algebras** (e.g. Breazu-Tannen/Meyer 1985):
 - Interpret **all terms** as partial functions
 - Substitution = composition, variables = projections
 - **Require** compatibility with deduction
- These are ‘the same’ as the natural categorical models — i.e. functors into partial cartesian closed categories (CSL 03)

Intensionality

Intensionality

Models are **intensional**

Intensionality

Models are **intensional**

- Allows e.g. topos models

Intensionality

Models are **intensional**

- Allows e.g. topos models
- Avoids problems such as incompleteness and non-existence of initial models

Intensionality

Models are **intensional**

- Allows e.g. topos models
- Avoids problems such as incompleteness and non-existence of initial models
- If desired, extensionality can be **specified**

Intensionality

Models are **intensional**

- Allows e.g. topos models
- Avoids problems such as incompleteness and non-existence of initial models
- If desired, extensionality can be **specified**
- **Internally**, everything is extensional (Mitchell/Scott 1989)

Type Classes and Polymorphism

Type Classes and Polymorphism

- **Type classes** can be declared or defined:

classes *Ord*; *Eq* < *Ord*; *Num* = { *a* : *Type* • *a* < *Int* }

Type Classes and Polymorphism

- **Type classes** can be declared or defined:
classes Ord ; $Eq < Ord$; $Num = \{a : Type \bullet a < Int\}$
- **Type constructors** may have classes in their **arities**:
var $a : Eq$
type $List\ a : Eq$

Type Classes and Polymorphism

- **Type classes** can be declared or defined:
classes Ord ; $Eq < Ord$; $Num = \{a : Type \bullet a < Int\}$
- **Type constructors** may have classes in their **arities**:
var $a : Eq$
type $List\ a : Eq$
- Operators and axioms may be **polymorphic** over classes:
var $a : Ord$
op $max : List\ a \rightarrow? a$

Semantics of Polymorphism

Semantics of Polymorphism

- **Classes** are subsets of the (syntactical) type universe

Semantics of Polymorphism

- **Classes** are subsets of the (syntactical) type universe
- Polymorphic types, operators, and axioms are at the first level coded by collections of instances
(second level: extension models, \rightarrow institution)

Semantics of Polymorphism

- **Classes** are subsets of the (syntactical) type universe
- Polymorphic types, operators, and axioms are at the first level coded by collections of instances (second level: extension models, \rightarrow institution)
- Axioms and operators **may** be ‘attached’ to classes to express proof obligations for instances:

```
class Ord { . . . } %% Order relation & axioms
```

```
type instance Nat
```

```
. . . %% Ordering on the naturals
```

Semantics of Polymorphism

- **Classes** are subsets of the (syntactical) type universe
- Polymorphic types, operators, and axioms are at the first level coded by collections of instances (second level: extension models, \rightarrow institution)
- Axioms and operators **may** be ‘attached’ to classes to express proof obligations for instances:

```
class Ord { . . . } %% Order relation & axioms
```

```
type instance Nat
```

```
. . . %% Ordering on the naturals
```

N.B.: System $F + \text{HOL}$ is known to be inconsistent!

The Internal Logic

The Internal Logic

- Specify internal equality

The Internal Logic

- Specify internal equality
- Define internal logic

The Internal Logic

- Specify internal equality
- Define internal logic ($(\forall x. \phi) = ((\lambda x. \phi) = \lambda x. tt)$ etc.)
- The internal logic is intuitionistic (essentially topos logic minus unique choice; codes dependent types)

The Internal Logic

- Specify internal equality
- Define internal logic ($(\forall x. \phi) = ((\lambda x. \phi) = \lambda x. tt)$ etc.)
- The internal logic is intuitionistic (essentially topos logic minus unique choice; codes dependent types)
- Extensionality implies classical logic!

The Internal Logic

- Specify internal equality
- Define internal logic ($((\forall x. \phi) = ((\lambda x. \phi) = \lambda x. tt)$ etc.)
- The internal logic is intuitionistic (essentially topos logic minus unique choice; codes dependent types)
- Extensionality implies classical logic!
- Datatypes: no junk/no confusion axioms in the internal logic

Recursion

Recursion

- HOLCF-like specification of cpo's (chain complete) as a type class

Recursion

- HOLCF-like specification of cpo's (chain complete) as a type class
- Continuous function spaces $s \xrightarrow{\text{cont}} t, s \xrightarrow{\text{cont}} ? t$

Recursion

- HOLCF-like specification of cpo's (chain complete) as a type class
- Continuous function spaces $s \xrightarrow{\text{cont}} t, s \xrightarrow{\text{cont}}? t$
- fixed point operator $Y : (a \rightarrow a) \rightarrow a$, where $a : Cppo$

Recursion: Syntax

Recursion: Syntax

- Standard functional programming syntax within **program** blocks

Recursion: Syntax

- Standard functional programming syntax within **program** blocks
- Abbreviate $f = Y(\lambda f \bullet \alpha)$ by

$$f\ x = \alpha\ x$$

Recursion: Syntax

- Standard functional programming syntax within **program** blocks
- Abbreviate $f = Y(\lambda f \bullet \alpha)$ by

$$f\ x = \alpha\ x$$

- Pattern syntax for recursive functions on datatypes, *let*-expressions, type inference

Recursion: Syntax

- Standard functional programming syntax within **program** blocks
- Abbreviate $f = Y(\lambda f \bullet \alpha)$ by

$$f\ x = \alpha\ x$$

- Pattern syntax for recursive functions on datatypes, *let*-expressions, type inference
- In short: **program** blocks ‘look and feel’ a lot like Haskell

Tool support

Tool support

- Implemented as part of HETS

Tool support

- Implemented as part of HETS
- Parser

Tool support

- Implemented as part of HETS
- Parser
- Static analysis: typing, mixfix analysis
(in progress: symbol maps; next: class analysis)

Tool support

- Implemented as part of HETS
- Parser
- Static analysis: typing, mixfix analysis
(in progress: symbol maps; next: class analysis)
- Encoding of `HASCASL` subset into Isabelle/HOL: in progress

Tool support

- Implemented as part of HETS
- Parser
- Static analysis: typing, mixfix analysis
(in progress: symbol maps; next: class analysis)
- Encoding of `HASCASL` subset into Isabelle/HOL: in progress
- Translation of executable subset into Haskell.

An Example

Watch this . . .

Conclusion

Conclusion

- HASCASL is conceptually simple

Conclusion

- HASCASL is conceptually simple
- . . . and yet accommodates both logic and programming.

Conclusion

- HASCASL is conceptually simple
- . . . and yet accommodates both logic and programming.
- Novel notion of semantics of the partial λ -calculus

Conclusion

- HASCASL is conceptually simple
- . . . and yet accommodates both logic and programming.
- Novel notion of semantics of the partial λ -calculus
- Programming features are **specified** within the language

Conclusion

- `HASCASL` is conceptually simple
- . . . and yet accommodates both logic and programming.
- Novel notion of semantics of the partial λ -calculus
- Programming features are **specified** within the language
- Executable `HASCASL` corresponds reasonably closely to Haskell

Future and 'Future' Work

Future and 'Future' Work

- Complete the tool support
- Specification methodology
- Case study
- Basic libraries

Future and 'Future' Work

- Complete the tool support
- Specification methodology
- Case study
- Basic libraries
- `HASCASL` for functional-imperative programming:
do-notation, monadic computational logics
(latest: computational logic with exceptions, AMAST 04)

The Global Element Construction

The Global Element Construction

$$\text{Cl}(\mathcal{T}) \xrightarrow{\text{ho}} (\mathbf{C}, \mathcal{M})$$

The Global Element Construction

$$\begin{array}{ccc} \text{Cl}(\mathcal{T}) & \xrightarrow{\text{ho}} & (\mathbf{C}, \mathcal{M}) \\ \downarrow \text{fo} & & \downarrow \text{hom}(1, -) \\ \text{Set} & \xlongequal{\quad} & \text{Set} \end{array}$$

The Global Element Construction

$$\begin{array}{ccc}
 \text{Cl}(\mathcal{T}) & \xrightarrow{\text{ho}} & (\mathbf{C}, \mathcal{M}) \\
 \downarrow \text{fo} & & \downarrow \text{hom}(1, -) \\
 \mathbf{Set} & \xlongequal{\quad} & \mathbf{Set}
 \end{array}$$

Process can be reversed: given first order $\text{Cl}(\mathcal{T}) \rightarrow \mathbf{Set}$, construct higher order $\text{Cl}(\mathcal{T}) \rightarrow (\mathbf{C}, \mathcal{M})$.

The Global Element Construction

$$\begin{array}{ccc}
 \text{Cl}(\mathcal{T}) & \xrightarrow{\text{ho}} & (\mathbf{C}, \mathcal{M}) \\
 \downarrow \text{fo} & & \downarrow \text{hom}(1, -) \\
 \mathbf{Set} & \xlongequal{\quad} & \mathbf{Set}
 \end{array}$$

Process can be reversed: given first order $\text{Cl}(\mathcal{T}) \rightarrow \mathbf{Set}$, construct higher order $\text{Cl}(\mathcal{T}) \rightarrow (\mathbf{C}, \mathcal{M})$.

(‘Henkin models demote higher order to first order’)