I will present research done under the supervision of Jan Schwinghammer and Prof. Gert Smolka, here at Saarland University.

I’m not going to get into detail, only give you the big picture.
The Big Picture
Our goal is to verify properties of object-oriented software systems, under more realistic assumptions and in a more modular way.

The strongest property we are interested in, is program correctness. And since we are dealing with infinite-state systems, with rich data structures and self-modifying code, we are using deductive verification.
Soundness

• Soundness of deduction systems is crucial
• Purely syntactic arguments
  • Suffice for type safety (subject-reduction)
  • Do not suffice for program correctness when executable code is first-class
  • Meaning of assertions no longer obvious
  • Assertions should describe code too
• Proving soundness wrt. semantic model:
  • “Derivability implies validity in the model”

The soundness of the employed deduction systems is crucial for deductive verification.

For proving soundness of type systems, syntactic arguments like subject-reduction suffice in general.

However, once code can be passed around or can be stored on the heap, and therefore it appears in assertions, purely syntactic arguments are not enough for program correctness. The meaning of assertions is no longer obvious, since they also have to describe code.

In this setting there is no escape, one has to prove soundness with respect to a semantic model, which makes a clear distinction between validity and derivability. A deduction system is sound if everything that can be derived using the rules is also valid with respect to the model.
The Challenge

• Hard to find good semantic models (in general)

• Even more challenging when:
  • Heap stores executable code
  • Realistic assumption (e.g. C/C++, Java, C#, ML)
  • Denotational semantics does not work!
    • Models extremely complex
    • Models not good enough
  • Possible alternative we are investigating:
    • “Step-indexing”

However, finding good semantic models in which one can reason about the behaviour of programs is in general hard. It is even more challenging when dynamic storage of executable code is allowed. But this feature is present in different forms in almost all practical programming languages: pointers to functions in C/C++, callbacks in Java and C# or general references in ML.

Classical denotational semantics just doesn't work in this setting! The models are so complex that it takes very smart people years to understand. And even with all this hard work, in the existing denotational models many operationally valid equations still fail.

Because of this, we are investigating a technique called "step-indexing", which can be used for building simpler semantic models.
Overview

- My Master’s thesis
  - Step-indexed semantic model
    - Soundness of expressive type system
- Further work
  - Local and modular programming logic for objects

For the future we plan to investigate the potential of step-indexed models as a basis for a program logics, while in my Master’s thesis I used a step-indexed model to prove soundness of a type system.

So, first I will present the results of my thesis, which I started in November, and I will submit in March, and then I will present the state of the art in program logics, and an idea on how we could improve on that.
In my thesis I studied the imperative object calculus – a simple but expressive object-oriented programming language. It is simple since only objects are considered as primitives. However, it is expressive enough to encode all common features of practical object-oriented languages like classes and inheritance.

The imperative object calculus captures the essence of object-oriented programming languages, the same way the lambda calculus captures the essence of functional ones.

And, what makes this calculus particularly interesting to us is that it features: dynamically allocated, higher-order store, so executable code can be dynamically stored on the heap.

Actually, we have also studied the simpler functional object calculus and have a similar model for it. However, in this talk I will only focus on the step-indexed model for the imperative object calculus.
Step-indexed Semantic Models

- First developed by Appel et al. - foundational PCC
- Machine-checkable proofs of type soundness
- Recursive types [Appel & McAllester, ‘01]
- General references and polymorphism [Ahmed et al., ‘03] [Ahmed, ‘04]
- We extended it with object types and subtyping

Step-indexed semantic models were first developed by Andrew Appel and his collaborators in the context of foundational proof-carrying code. Their goal is having machine-checkable proofs of type soundness, for low-level languages, like assembler or machine code.

However, they also defined a step-indexed semantic model for a lambda calculus with recursive types, which then Amal Ahmed successfully extended to general references and polymorphism.

We have further extended this model with object types and subtyping, in order to use it for our object calculus.
Semantic Types

- Use only operational semantics of untyped calculus
- Types are sets of indexed values
  \[ a : k \alpha \text{ if } a \text{ behaves like an element of } \alpha \text{ for } k \text{ steps} \]
- Types have a “meaning”
  - Predicates over programs
  - Type constructors
    - Functions over these predicates
- Simple models - set-theoretic
- Simple proofs - induction on naturals

Step-indexed semantic models are based only on the small-step operational semantics of an untyped calculus.

Types are then interpreted as sets of indexed values. Informally, an expression has a certain type if it behaves like an element of that type for a fixed number of steps. Or equivalently we say that \( a \) is in \( \alpha \) with approximation \( k \), if one cannot distinguish \( a \) from a real value of type \( \alpha \) in less than \( k \) computation steps.

We call these semantic types, because like in a denotational semantics types have a meaning: they can be viewed as predicates over programs, while type constructors are just functions over these predicates. However, these models are purely set-theoretic, so much simpler than the models one usually has with a denotational semantics. And the proofs are usually just by direct induction on the index, so also simple.
Type System

- Very expressive
  - Object types \([m_d : \tau_d]_{d \in D}\)
  - Recursive types \(\mu F\)
  - Polymorphic types \(\forall F \exists F\)
  - Subtyping \(\alpha \subseteq \beta\)
    - Very natural - just set inclusion
    - Bounded quantification \(\forall \tau \subseteq \alpha\)
    - Self types \(\sim\) recursive object types

The type system we considered is very expressive. It has of course object types, but also recursive and polymorphic types. And what is very important in this setting, is has subtyping.

Actually our model has a very natural notion of subtyping: since types are just sets, subtyping is just set inclusion. And while defining subtyping is very simple, the interaction between subtyping and the other features of the type system is not trivial. This interaction also gives raise to new types, like bounded polymorphic types and self types (recursive object types with proper subtyping).
Type Safety and Soundness

- Type safety = “well typed terms don’t get stuck”
- Immediate (by construction)
- Typing rules need to be proved sound
  - 28 semantic typing rules
    (+32 rules for functional object calculus)
- Each rule proved in isolation - modularity
- Foundational soundness proofs
  - Machine checkable (ours not checked yet)

In our setting, type safety (the fact that “well typed terms don’t get stuck”) is immediate from the construction of our model, while the usual typing rules needed to be proved sound with respect to the model.

However, the fact that each rule is proved in isolation gives us more modularity, compared to subject-reduction.

Appel and his collaborators have also shown that such foundational proofs can be encoded into an expressive logic (HOL), and then machine checked. (For us this was not crucial, so for now we did not check our proofs automatically.)
Contributions

- Semantic model for the imperative object calculus
  - Extends step-indexed model of [Ahmed ‘04]
    - Object types
    - Subtyping
- First soundness proof of expressive type system

The contributions of my thesis are successfully creating a semantic model for the imperative object calculus and using it to provide the first soundness proof of an expressive type system.

[Compared to the existing step-indexed semantic models]
New in our model are the object types and the treatment of subtyping.
Further Work

- Imperative object calculus
- **Local and modular programming logic**
- Object invariants
- Object encapsulation and ownership
- Program equivalence
- Information flow security
- Concurrent objects
- Security properties (e.g. secrecy, authentication)
- Transferring results to real languages (e.g. Java)

An interesting direction for further work, would be to extend our result to a local and modular programming logic for the imperative object calculus. But before I go into some details about this, I have to mention that there are also other open problems I find interesting, some of them closely related to my current work, others not.

Imperative object calculus
- Object invariants
- Object encapsulation and ownership
- Operationally reasoning about program equivalence (there has been a lot of progress in recent years in reasoning about program equivalence based on operational semantics)
- Information flow security (non-interference)

Concurrent objects
- asserting atomicity, determinism, deadlock–freedom, thread–safe data structures, etc.
- security properties like secrecy and authentication – from spi calculus

Transferring results to real languages (Java)
Hoare Logic was surely not local nor modular. Actually specification logic was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean compositionality. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

Locality or separation is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular. Actually **specification logic** was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean **compositionality**. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

**Locality** or **separation** is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular. Actually specification logic was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean compositionality. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

Locality or separation is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular. Actually **specification logic** was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean **compositionality**. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

**Locality** or **separation** is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular.
Actually specification logic was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean compositionality. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

Locality or separation is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular. Actually specification logic was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean compositionality. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

Locality or separation is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular. Actually **specification logic** was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean **compositionality**. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification. **Locality** or **separation** is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
Hoare Logic was surely not local nor modular.
Actually **specification logic** was the first modular program logic. It allowed modules to be developed and proved correct independently, while guaranteeing the correctness of all their possible compositions. So by modularity, I mean **compositionality**. For example, if you have the specification of an abstract data structure, and you prove your client code correct with respect to it, then your will know that your client code will work correctly with any implementation that satisfies the same specification.

**Locality** or **separation** is an orthogonal concern. It is very common that your operations do not change state, or change only a small, well-defined part of the state, while everything else stays the same. In Hoare logic, you have to explicitly specify and prove that an operation does not the change existing state. In separation logic specifications are tight by default, meaning that your assertions only need to describe the things that change.

What’s also important to note, is that almost all recent results in program logics focus on local and modular reasoning. The setting in which program logics are devised can however differ.
The **setting** in which program logics are devised can however differ. Much work was done for **procedural languages**, usually **without objects**, and definitely without higher-order store.

For **separation and higher-order store the only results** are by my supervisor for a very simple language of commands, and then there are two papers which are going to be published this year in the setting of monadic lambda calculi.

So it’s natural to ask: can’t we have a logic for objects and higher-order store?
The setting in which program logics are devised can however differ. Much work was done for procedural languages, usually without objects, and definitely without higher-order store.

For separation and higher-order store the only results are by my supervisor for a very simple language of commands, and then there are two papers which are going to be published this year in the setting of monadic lambda calculi.

So it’s natural to ask: can’t we have a logic for objects and higher-order store?
There is one such logic, introduced by Abadi and Leino, and later proved sound by my supervisor (using a denotational model). However, this logic is not local and not modular – it’s basically Hoare logic for objects.

So the question now is: Can we do better? Can we have a local and modular program logic for the imperative object calculus?
There is one such logic, introduced by Abadi and Leino, and later proved sound by my supervisor (using a denotational model). However, this logic is not local and not modular – it’s basically Hoare logic for objects.

So the question now is: Can we do better? Can we have a local and modular program logic for the imperative object calculus?
There is one such logic, introduced by Abadi and Leino, and later proved sound by my supervisor (using a denotational model). However, this logic is not local and not modular – it's basically Hoare logic for objects.

So the question now is: Can we do better? Can we have a local and modular program logic for the imperative object calculus?
Summary

• Imperative object calculus
• Step-indexed semantic model
• First soundness proof of expressive type system
• Further research
  • Local and modular program logic

To sum things up, I have presented the results of my master’s thesis, where I constructed a step-indexed semantic model for the imperative object calculus, which I used to prove the soundness of an expressive type system.

And, I have presented one interesting direction for further research: constructing a local and modular program logic for the imperative object calculus.
Imperative Object Calculus

\[ a, b ::= \begin{array}{l}
  x & \text{variables} \\
  \{ m_d = l_d \}_{d \in D} & \text{object value} \\
  [ m_d = \zeta(x_d)b_d ]_{d \in D} & \text{object creation} \\
  \text{clone } a & \text{object cloning} \\
  a.m & \text{method invocation} \\
  a.m := \zeta(x)b & \text{method update} \\
  \lambda x. b & \text{procedures} \\
  a \ b & \text{procedure application} \\
  \Lambda a & \text{type abstraction} \\
  a [] & \text{type application} \\
  \text{pack } a & \text{creating existential package} \\
  \text{open } a \text{ as } x \text{ in } b & \text{opening package}
\end{array} \]
Example Programs

- **Factorial**
  \[
  \text{fac} = \varsigma(y)\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times y.\text{fac}(n - 1)
  \]

- **Euclid's GCD Algorithm**
  \[
  \text{gcd} = \varsigma(y)\lambda x. \lambda z. \text{if } x < z \text{ then } y.\text{gcd} x (z - x)
  \]
  \[
  \text{else if } z < x \text{ then } y.\text{gcd} (x - z) z \text{ else } x
  \]
Example Expression: Type

\[ f = \varsigma(x)[] : 1 \ [f : int] \]
Semantic vs. Syntactic Rules

(PACK) \[ \exists \tau \in \text{Type. } \tau \subseteq \alpha \land \Gamma \models a : F(\tau) \]
\[ \Gamma \models \text{pack } a : \exists \alpha F \]

(SYN-PACK) \[ \Psi, \Gamma \vdash C \leq A \]
\[ \Psi, \Gamma \vdash a[X := C] : B[X := C] \]
\[ \Psi, \Gamma \vdash \text{pack } X \leq A = C \text{ a : } \exists X \leq A. B \]

- Semantic rules
  - Type variables are not tracked
  - We only check location-free programs
- Syntactic rules
  - Efficiently decidable
Values and Programs

\[ v \in \text{Val} ::= \{ m_d = l_d \}_{d \in D} \mid \lambda x. b \mid \Lambda a \mid \text{pack } v \]

\[ \text{CVal} = \{ v \in \text{Val} \mid FV(v) = \emptyset \} \]

\[ \text{Prog} = \{ v \in \text{CVal} \mid Labels(v) = \emptyset \} \]
Operational Semantics

\[ \forall d \in D. \ l_d \not\in \text{Dom}(S) \]

\[ \langle S, [m_d = \varsigma(x_d)b_d]_{d \in D} \rangle \rightarrow \langle S[l_d \mapsto \lambda x_d. b_d]_{d \in D}, \{m_d = l_d\}_{d \in D} \rangle \]

\[ \forall d \in D. \ l'_d \not\in \text{Dom}(S) \]

\[ \langle S, \text{clone} \{m_d = l_d\}_{d \in D} \rangle \rightarrow \langle S[l'_d \mapsto S(l_d)]_{d \in D}, \{m_d = l'_d\}_{d \in D} \rangle \]

\[ e \in D \]

\[ \langle S, \{m_d = l_d\}_{d \in D}.m_e \rangle \rightarrow \langle S, S(l_e) \{m_d = l_d\}_{d \in D} \rangle \]

\[ e \in D \]

\[ \langle S, \{m_d = l_d\}_{d \in D}.m_e := \varsigma(x)b \rangle \rightarrow \langle S[l_e \mapsto \lambda x. b], \{m_d = l_d\}_{d \in D} \rangle \]

\[ \langle S, (\lambda x. b) v \rangle \rightarrow \langle S, [x \mapsto v](b) \rangle \]

\[ \langle S, a \rangle \rightarrow \langle S', b \rangle \]

\[ \langle S, C[a] \rangle \rightarrow \langle S', C[b] \rangle \]

\[ C[\bullet] = \bullet \mid C.m \mid C.m := \varsigma(x)b \mid C \ b \mid v \ C \mid C \ [] \mid \text{pack } C \mid \text{open } C \text{ as } x \text{ in } b \]
Step-indexed Semantic Model

Safety

$$\text{Safe}_k = \{(S, a) \mid \forall j < k. \forall S', b. \langle S, a \rangle \rightarrow^j \langle S', b \rangle \Rightarrow b \in \text{Val} \lor \neg \text{irred}(S', b)\}$$

$$\text{Safe} = \bigcap_{k \geq 0} \text{Safe}_k$$

Circular “definition”

$$\text{PreType} = \mathcal{P}(\mathbb{N} \times \text{StoreType} \times \text{Val})$$

$$\text{StoreType} = \text{Loc} \rightarrow_{\text{fin}} \text{PreType}$$

Approximation

$$[\tau]_k = \{\langle j, \Psi, v \rangle \in \tau \mid j < k\}$$

$$[\Psi]_k = \lambda l \in \text{Loc.} \ [\Psi(l)]_k$$
Stratification

PreType_0 = \{\emptyset\}

PreType_{k+1} = \{\tau \mid \forall \langle j, \Psi, v \rangle \in \tau. \ j \leq k \ \land \ \Psi \in \text{StoreType}_j\}\}

StoreType_k = \{\Psi \mid \forall l \in \text{Dom}(\Psi). \ \Psi(l) \in \text{Type}_k\}\}

PreType = \{\tau \mid \forall k \geq 0. \ [\tau]_k \in \text{PreType}_k\}\}

StoryType = \{\Psi \mid \forall k \geq 0. \ [\Psi]_k \in \text{StoreType}_k\}\}
Semantic Types

State Extension

\[(k, \Psi) \sqsubseteq (j, \Psi') : \iff j \leq k \land \forall l \in Dom(\Psi). [\Psi']_j(l) = [\Psi]_j(l)\]

Types

\textit{PreTypes} closed under state extension.

\[
\text{Type} = \{ \tau \in \text{PreType} \mid \forall k, j \geq 0. \forall \Psi, \Psi'. \forall v \in \text{Val}.
\quad ((k, \Psi) \sqsubseteq (j, \Psi') \land \langle k, \Psi, v \rangle \in \tau) \Rightarrow \langle j, \Psi', v \rangle \in \tau \}\]
Well-typed Store \((S :_k \Psi)\)

\[
(S :_k \Psi : \iff \text{Dom}(\Psi) \subseteq \text{Dom}(S) \land \forall j < k. \forall l \in \text{Dom}(\Psi). \langle j, [\Psi]_j, S(l) \rangle \in [\Psi]_k(l)
\]

Expr\(:_{k,\Psi} \text{Type}\)

\[
a :_{k,\Psi} \tau : \iff \text{FV}(a) = \emptyset \land \forall j < k, S, S', b.
(S :_k \Psi \land \langle S, a \rangle \rightarrow^j \langle S', b \rangle \land \text{irred}(S', b)) \Rightarrow
\exists \Psi'. (k, \Psi) \sqsubseteq (k - j, \Psi') \land S' :_{k - j} \Psi' \land \langle k - j, \Psi', b \rangle \in \tau
\]

Value Env.\(:_{k,\Psi} \text{Type Env.}\)

\[
\Gamma : \text{Var} \rightarrow_{\text{fin}} \text{Type} \text{ (type environment)}
\]

\[
\gamma : \text{Var} \rightarrow_{\text{fin}} \text{CVal} \text{ (value environment)}
\]

\[
\gamma :_{k,\Psi} \Gamma : \iff \forall x \in \text{Dom}(\Gamma). \gamma(x) :_{k,\Psi} \Gamma(x)
\]
Semantic Type Judgement

Let a such that $Labels(a) = \emptyset$.

\[ \Gamma \vdash^k a : \alpha \iff FV(a) \subseteq Dom(\Gamma) \]
\[ \land \forall \Psi. \forall \gamma : k, \Psi \Gamma. \gamma(a) : k, \Psi \alpha \]
\[ \Gamma \vdash a : \alpha \iff \forall k \geq 0. \Gamma \vdash^k a : \alpha \]
\[ \vdash a : \alpha \iff \emptyset \vdash a : \alpha \]

Properties

1. $j \leq k \Rightarrow (k, \Psi) \sqsubseteq (j, [\Psi]_j)$
2. $(j \leq k \land a : k, \Psi \alpha) \Rightarrow a : j, [\Psi]_j \alpha$
3. $\forall v \in Val. v : k, \Psi \tau \Rightarrow \langle k, \Psi, v \rangle \in \tau$
4. $\vdash a : \alpha \Rightarrow \forall S. (S, a) \in Safe$ (type soundness)
Definitions of Types

$\bot = \emptyset$

$\top = \{ \langle k, \Psi, v \rangle \mid k \in \mathbb{N}, \Psi \in \text{Store Type}, v \in \text{Val} \}$

$\alpha \rightarrow \beta = \{ \langle k, \Psi, \lambda x. b \rangle \mid \forall j < k. \forall \Psi, v \in \text{Val}. \ ( (k, \Psi) \sqsubseteq (j, \Psi') \land \langle j, \Psi', v \rangle \in \alpha) \Rightarrow [x \mapsto v](b) : j, \Psi', \tau \}$

$\alpha = [m_d : \tau_d]_{d \in D} = \{ \langle k, \Psi, \{ m_e = l_e \}_{e \in E} \rangle \mid D \subseteq E \land \exists \alpha'. [\alpha']_k \in \text{Type} \land [\alpha']_k \subseteq [\alpha]_k \land \forall j < k. \langle j, \Psi, \{ m_e = l_e \}_{e \in E} \rangle \in \alpha' \land \forall d \in D. [\Psi]_{k}(l_d) = [\alpha' \rightarrow \tau_d]_k \}$

$\mu F = \{ \langle k, \Psi, v \rangle \mid \langle k, \Psi, v \rangle \in F^{k+1}(\bot) \}$

$\forall \alpha F = \{ \langle k, \Psi, \Lambda.a \rangle \mid \forall j, \Psi'. \forall \tau. (k, \Psi) \sqsubseteq (j, \Psi') \land [\tau]_j \in \text{Type} \land [\tau]_j \subseteq [\alpha]_j \Rightarrow \forall i < j. a : i, [\Psi']_i F(\tau) \}$

$\exists \alpha F = \{ \langle k, \Psi, \text{pack} v \rangle \mid \exists \tau. [\tau]_k \in \text{Type} \land [\tau]_k \subseteq [\alpha]_k \land \forall j < k. \langle j, [\Psi]_j, v \rangle \in F(\tau) \}$
Let \( \alpha = [m_d : \tau_d]_{d \in D} \)

**OBJ**
\[
\forall d \in D. \Gamma[x_d : \alpha] \models b_d : \tau_d \\
\Gamma \models [m_d = \varsigma(x_d)b_d]_{d \in D} : \alpha
\]

**SEL**
\[
\Gamma \models a : \alpha \quad e \in D \\
\Gamma \models a.m_e : \tau_e
\]

**UPD**
\[
\Gamma \models a : \alpha \quad e \in D \quad \Gamma[x : \alpha] \models b : \tau_e \\
\Gamma \models a.m_e := \varsigma(x)b : \alpha
\]
Recursive Types

\[
\begin{align*}
\text{(UNFOLD)} & \quad \frac{\Gamma \vdash a : \mu F}{\Gamma \vdash a : F(\mu F)} \\
\text{(FOLD)} & \quad \frac{\Gamma \vdash a : F(\mu F) \quad F \text{ contractive}}{\Gamma \vdash a : \mu F}
\end{align*}
\]
Polymorphic Types

\((\text{TABS})\)
\[
\forall \tau \in \text{Type}. \ \tau \subseteq \alpha \Rightarrow \Gamma \models a : F(\tau)
\]
\[
\Gamma \models \Lambda. a : \forall_{\alpha} F
\]

\((\text{TAPP})\)
\[
\Gamma \models a : \forall_{\alpha} F \quad \tau \in \text{Type} \quad \tau \subseteq \alpha
\]
\[
\Gamma \models a [\cdot] : F(\tau)
\]

\((\text{PACK})\)
\[
\exists \tau \in \text{Type}. \ \tau \subseteq \alpha \land \Gamma \models a : F(\tau)
\]
\[
\Gamma \models \text{pack } a : \exists_{\alpha} F
\]

\((\text{OPEN})\)
\[
\Gamma \models a : \exists_{\alpha} F \quad \forall \tau \in \text{Type}. \ \tau \subseteq \alpha \Rightarrow \Gamma[x : F(\tau)] \models b : \beta
\]
\[
\Gamma \models \text{open } a \text{ as } x \text{ in } b : \beta
\]
Subtyping

\[(\text{SubSub}) \quad \frac{\Gamma \models a : \alpha \quad \alpha \subseteq \beta}{\Gamma \models a : \beta}\]

\[(\text{Sem-SRefl}) \quad \alpha \subseteq \alpha \quad (\text{SubTrans}) \quad \frac{\alpha \subseteq \tau \quad \tau \subseteq \beta}{\alpha \subseteq \beta}\]

\[(\text{SubTop}) \quad \alpha \subseteq \top \quad (\text{SubBot}) \quad \bot \subseteq \alpha\]
Subtyping

(SubObj) \[ E \subseteq D \]
\[
[m_d : s(x_d) \tau_d]_{d \in D} \subseteq [m_e : s(x_e) \tau_e]_{e \in E}
\]

(SubRec) \[
\forall \alpha, \beta \in \text{Type. } \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq G(\beta)
\]
\[
\mu F \subseteq \mu G
\]

(SubUniv) \[
\beta \subseteq \alpha \quad \forall \tau \in \text{Type. } \tau \subseteq \beta \Rightarrow F(\tau) \subseteq G(\tau)
\]
\[
\forall \alpha F \subseteq \forall \beta G
\]

(SubExist) \[
\alpha \subseteq \beta \quad \forall \tau \in \text{Type. } \tau \subseteq \alpha \Rightarrow F(\tau) \subseteq G(\tau)
\]
\[
\exists \alpha F \subseteq \exists \beta G
\]
$sF = \mu(\lambda \alpha \in \text{Type}. \exists \alpha F)$

(WRAP) \[\exists \tau \in \text{Type}. \tau \subseteq sF \land \Gamma \vdash a : F(\tau)\]
\[\Gamma \vdash \text{pack } a : sF\]

(USE) \[\Gamma \vdash a : sF \quad \forall \tau \in \text{Type. } \tau \subseteq sF \Rightarrow \Gamma[x : F(\tau)] \vdash b : \beta\]
\[\Gamma \vdash \text{open } a \text{ as } x \text{ in } b : \beta\]

(SUBSELF) \[\forall \tau \in \text{Type. } F(\tau) \subseteq G(\tau)\]
\[sF \subseteq sG\]
Self Types

Self object creation: pack \([m_d = \varsigma(x_d)b_d]_{d \in D}\)

Selection: \(a \cdot m_d \equiv \) open \(a\) as \(x\) in \(x.m_d\)

Update: \(a \cdot m_d := \varsigma(x)b \equiv \) open \(a\) as \(x\) in pack \((x.m_d := \varsigma(x)b)\)

Let \(\alpha = \varsigma(\lambda \tau \in \text{Type.} [m_d : F_d(\tau)]_{d \in D})\)

\[\text{(SELFOBJ)} \quad \exists \tau \in \text{Type.} \tau \subseteq \alpha \land \forall d \in D. \Gamma[x_d : \tau] \models b_d : F_d(\tau) \]

\[\Gamma \models \text{pack} [m_d = \varsigma(x_d)b_d]_{d \in D} : \alpha\]

\[\text{(SELFSEL)} \quad \Gamma \models a : \alpha \quad e \in D \]

\[\Gamma \models a \cdot m_e : F_e(\alpha)\]

\[\text{(SELFUPD)} \quad \forall \tau \in \text{Type.} \tau \subseteq \alpha \Rightarrow \Gamma[x : [m_d : F_d(\tau)]_{d \in D}] \models b : F_e(\tau) \]

\[\Gamma \models a \cdot m_e := \varsigma(x)b : \alpha\]

\[\text{(SUBSELFOBJ)} \quad E \subseteq D \quad \varsigma(\lambda \tau \in \text{Type.} [m_d : F_d(\tau)]_{d \in D}) \subseteq \varsigma(\lambda \tau \in \text{Type.} [m_e : F_e(\tau)]_{e \in E})\]
Notations

\(a, b \in \text{Ter}\) terms
\(d \in D, e \in E\) iterators over some finite sets
\(f, g, h \in F\) field names
\(i, j, k \in \mathbb{N}\) indices (usually \(i < j < k\))
\(l \in \text{Loc}\) locations
\(m \in M\) method names
\(u, v, w \in \text{Val}\) values
\(x, y, z \in \text{Var}\) variables
\(C\) evaluation context
\(F, G, H\) functionals
\(S\) store
\(\alpha, \beta, \tau\) semantic types
\(\sigma, \gamma\) substitutions (\(\gamma\) is “ground”)
\(\Psi\) store typing
\(\Gamma\) type environment