TinkerType:

A Framework for Playing with Formal Systems

Benjamin C. Pierce
Michael Levin
University of Pennsylvania

The Setting

Two type systems good

$\Longrightarrow$

Four type systems better!

The Problem

But...

> Experimenting with new type systems is hard!

There are lots of reasons for this. One is that there is too much repeated work: each new experiment starts from scratch.

The Problem

But...

Experimenting with new type systems is hard!

There are lots of reasons for this. One is that there is too much repeated work: each new experiment starts from scratch.

Need software engineering techniques for type systems...

> "How can we reuse what we've already done?"

Classical approach: Parameterize

Pure Type Systems

Advantages:

- ✔ insight
- ✔ once-and-for-all proofs of useful properties

Classical approach: Parameterize

Pure Type Systems

Advantages:

- ✔ insight
- ✔ once-and-for-all proofs of useful properties

Disadvantages:

- ✘ technical (requires deep understanding of common structures)
- ✘ pedagogical (ever looked at the simply typed lambda-calculus as a PTS?)

Alternative approach: Inherit

Parameterization: Define a family of structures by giving a single parametric structure, with parameters for all the varying parts.

Inheritance: Start with simplest common structure. Define more complex structures by deltas from simpler ones.

Alternative approach: Inherit

Parameterization: Define a family of structures by giving a single parametric structure, with parameters for all the varying parts.

Inheritance: Start with simplest common structure. Define more complex structures by deltas from simpler ones.

(Cf. functional vs. object-oriented programming styles)

Granularity of Inheritance

Inheritance is used to define new systems from old ones by copying some “chunks” of functionality and overriding others.

Key problem: breaking down functionality into chunks that are:

- reasonably independent
- a convenient size

Granularity of Inheritance

Inheritance is used to define new systems from old ones by copying some “chunks” of functionality and overriding others.

Key problem: breaking down functionality into chunks that are:

- reasonably independent
- a convenient size

For type systems (and similar formal systems based on inductively defined relations over trees), inference rules offer a natural unit of inheritance / overriding.

TinkerType

The basic idea:

- take a family of type systems
- cut them up into individual clauses (inference rules)
- annotate each clause with a collection of features that characterize the systems it belongs in
- identify different versions of the same rule by giving them the same name
- given a set of features, build a system by extracting the most specific rules tagged with these features.

TinkerType

The basic idea:

- take a family of type systems
- cut them up into individual clauses (inference rules)
- annotate each clause with a collection of features that characterize the systems it belongs in
- identify different versions of the same rule by giving them the same name
- given a set of features, build a system by extracting the most specific rules tagged with these features.

Note that clauses, features, and names are uninterpreted by the formalism.

Example

Simply typed lambda-calculus:

$$\frac{\Gamma \vdash t_1 \in T_2 \rightarrow T_1 \qquad \Gamma \vdash t_2 \in T_2}{\Gamma \vdash t_1\ t_2 \in T_1} \qquad \text{T-App [arrow, typing]}$$

Example

Simply typed lambda-calculus:

$$\frac{\Gamma \vdash t_1 \in T_2 \to T_1 \qquad \Gamma \vdash t_2 \in T_2}{\Gamma \vdash t_1\ t_2 \in T_1}$$

T-App [arrow, typing]

Simply typed lambda-calculus with subtyping:

$$\frac{\Gamma \vdash t_1 \in T_2 \to T_1 \qquad \Gamma \vdash t_2 \in U \qquad U <: T_2}{\Gamma \vdash t_1\ t_2 \in T_1}$$

T-App [arrow, sub, typing]

Refinement: Feature dependencies

Every system with a subtyping judgement also contains a typing judgement. We record this once and for all by introducing a feature dependency of sub on typing. We may then drop the typing annotation where it is implied:

$$\frac{\Gamma \vdash t_1 \in T_2 \to T_1 \qquad \Gamma \vdash t_2 \in U \qquad U <: T_2}{\Gamma \vdash t_1\ t_2 \in T_1} \quad \text{T-APP [arrow, sub]}$$

In general, feature dependency is a relation between sets of features and sets of features. (E.g., the pair of features [bool, sub] depends on the feature calcjoins in algorithmic systems with subtyping.)

Refinement: Feature constraints

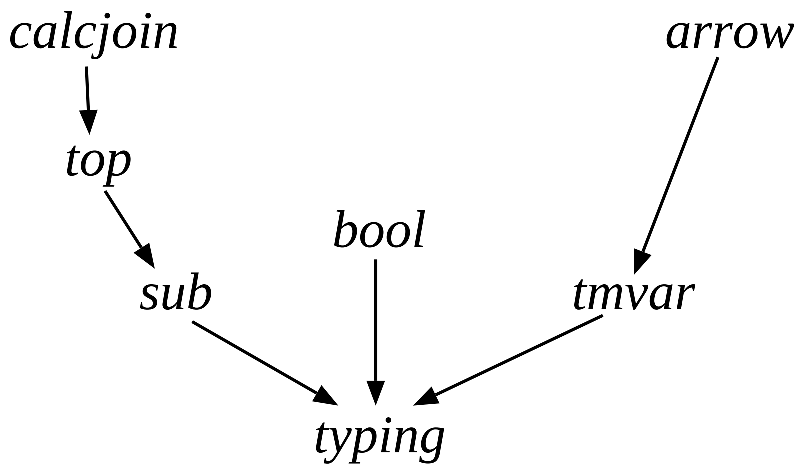
It is sometimes useful to introduce more general constraints between features. E.g.:

> Any type system whose syntax includes recursive types must either provide explicit `fold`/`unfold` isomorphisms or else be equipped with extra type equivalence rules for performing unfolding automatically.

TinkerType supports arbitrary propositional formulas whose atoms are features. Implicational formulas are treated specially: given a set of features describing a system, we

1. close under the implicational formulas
2. check that the rest of the formulas are satisfied.

Features for arrows, booleans, and subtyping



Plus: bool $\wedge$ sub $\Longrightarrow$ calcjoin

Complete example

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} \in \mathtt{T}}$$

T-VAR [tmvar, typing]

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_2 \vdash \mathtt{t}_1 \in \mathtt{T}_1}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_1 \in \mathtt{T}_2 {\rightarrow} \mathtt{T}_1}$$

T-LAM [arrow, typing]

$$\Gamma \vdash \mathtt{false} \in \mathtt{Bool}$$

T-FALSE [bool, typing]

$$\Gamma \vdash \mathtt{true} \in \mathtt{Bool}$$

T-TRUE [bool, typing]

$$\frac{\Gamma \vdash t_1 \in T_2 \to T_1 \qquad \Gamma \vdash t_2 \in T_2}{\Gamma \vdash t_1\ t_2 \in T_1}$$

T-App [arrow, typing]

$$\frac{\Gamma \vdash t_1 \in T_2 \to T_1 \qquad \Gamma \vdash t_2 \in U \qquad U <: T_2}{\Gamma \vdash t_1\ t_2 \in T_1}$$

T-App [arrow, sub]

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

S-Arrow [arrow, sub]

$$S <: \mathtt{Top}$$

S-Top [top]

$$\mathtt{Bool} <: \mathtt{Bool}$$

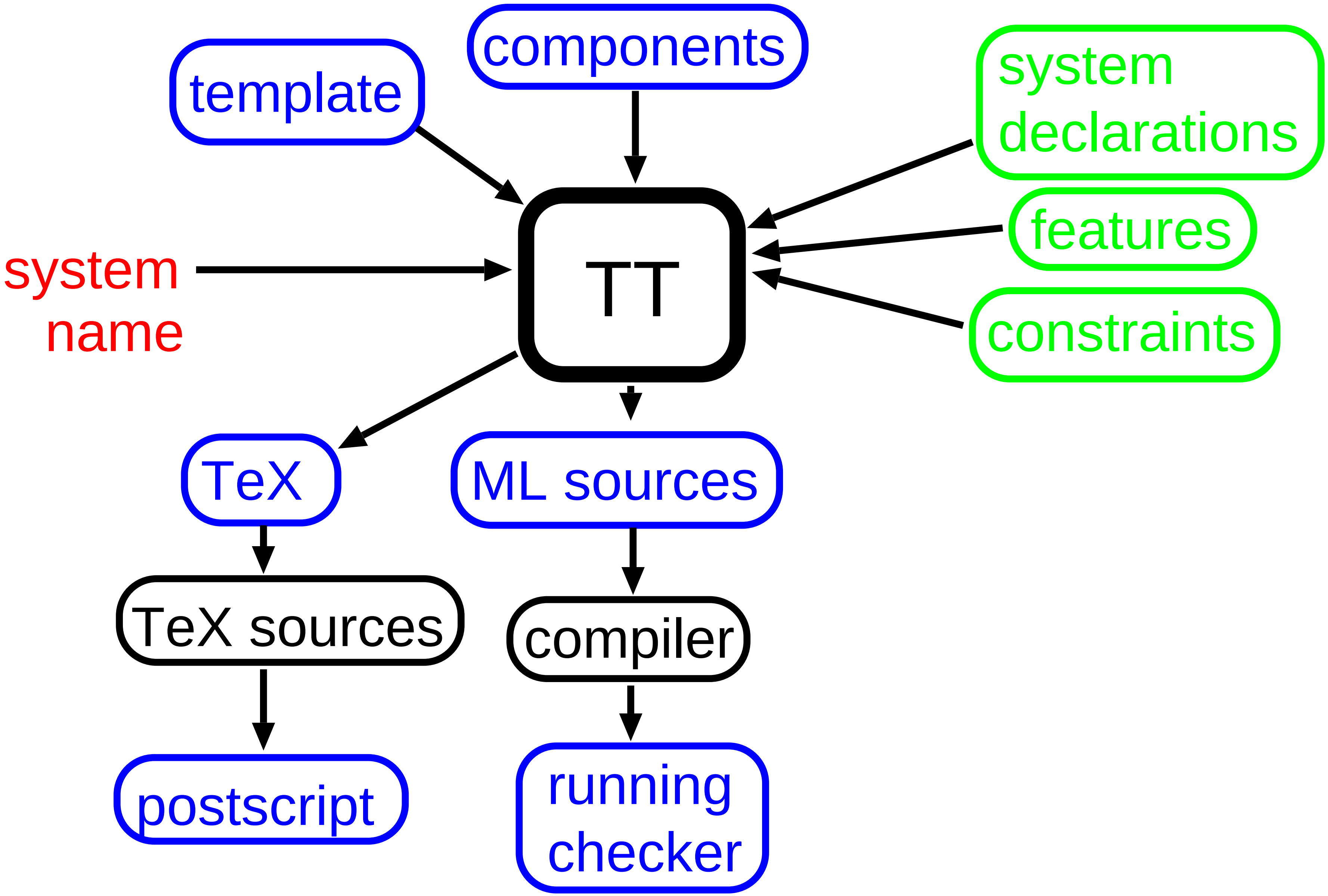
S-Bool [bool, sub]

$$\frac{\Gamma \vdash \mathtt{t}_1 \in \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 \in \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 \in \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \in \mathtt{T}} \text{ T-IF [bool, typing]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 \in \mathtt{T}_1 \qquad \mathtt{T}_1 <: \mathtt{Bool} \\ \Gamma \vdash \mathtt{t}_2 \in \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_3 \in \mathtt{T}_3 \qquad \mathtt{T} = \mathtt{T}_2 \vee \mathtt{T}_3\end{array}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \in \mathtt{T}} \text{ T-IF [bool, calcjoin]}$$

Assembling Systems

To build a system, we specify a set of features plus a file or directory, called a template, containing a skeleton for the generated system. For ML systems, the template is a directory containing a makefile, boilerplate for lexing and parsing, a top-level command loop, and skeleton modules (containing holes marked with the names of sections to be inserted at each point) for the major components of the typechecker.


template
components
system declarations
system name
TT
features
constraints
TeX
ML sources
TeX sources
compiler
postscript
running checker

The TinkerType Assembler

- Structuring aids
  - “Components” for grouping clauses with similar features
  - Sections / subsections
- Consistency checking
- Ordering constraints
- Control over appearance
  - Headers / footers / separators
  - Pretty printing
  - Coloring new rules / parts of rules...
  - Selective elision

Demo...

Current status

- TinkerType tool: ~3,000 lines of ML
- Component library: ~12,000 lines
  - 20% TeX clauses
  - 60% ML clauses
  - 20% tinkertype proper
- Currently generating ~70 distinct typecheckers, totalling ~140,000 lines of ML (plus ~4000 lines of TeX)

Future

- The basic TinkerType framework should be applicable to modular presentations of a wide variety of formal systems

  Basically, any family of systems that can naturally be described as sets of mutually inductive relations on trees...
  - type systems for things other than lambda-calculi (e.g. Abadi-Cardelli calculi)
  - wide variety of logics
  - other aspects of programming languages?
  - proofs of essential metatheoretical properties???!
- More static checking
  - “Judgement signatures”

Future, cont'd

What about modular proofs?

Future, cont'd

What about modular proofs?

Not much hope of making TinkerType do proofs, since it doesn't understand anything about the structures it manipulates.

Future, cont'd

What about modular proofs?

Not much hope of making TinkerType do proofs, since it doesn't understand anything about the structures it manipulates.

But it does seem plausible to make TinkerType generate proof scripts that could then be verified with your favorite proof checker.

(After all, proofs are nothing but collections of inductively defined relations on trees...)

Future, cont'd

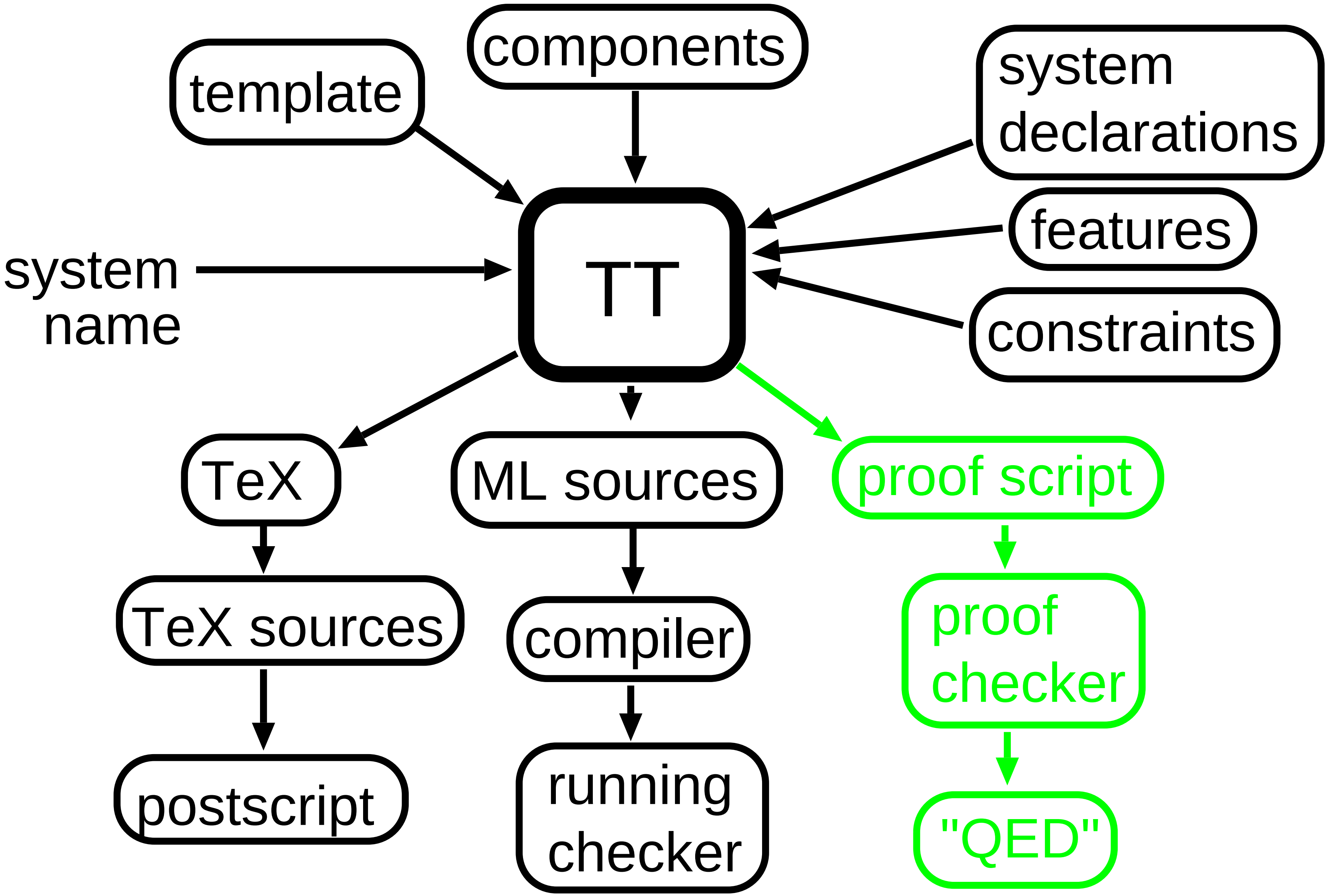
What about modular proofs?

Not much hope of making TinkerType do proofs, since it doesn't understand anything about the structures it manipulates.

But it does seem plausible to make TinkerType generate proof scripts that could then be verified with your favorite proof checker.

(After all, proofs are nothing but collections of inductively defined relations on trees...)

A small feasibility study (modular proofs of subject reduction for $F_{<:}$ and its subsystems, generated in English and hand-checked) yielded promising results...


template
components
system declarations
system name
TT
features
constraints
TeX
ML sources
proof script
TeX sources
compiler
proof checker
postscript
running checker
"QED"

Evaluation (positive)

- ✔ We succeeded in developing a substantial library of type systems

Evaluation (positive)

- ✔ We succeeded in developing a substantial library of type systems
- ✔ The features we used constitute an interesting taxonomy of type systems

Evaluation (positive)

- ✔ We succeeded in developing a substantial library of type systems
- ✔ The features we used constitute an interesting taxonomy of type systems
- ✔ The basic concepts of features and clauses feel quite natural in practice

Evaluation (positive)

- ✔ We succeeded in developing a substantial library of type systems
- ✔ The features we used constitute an interesting taxonomy of type systems
- ✔ The basic concepts of features and clauses feel quite natural in practice
- ✔ The implementation is cool

Evaluation (negative)

- ✘ More structure needed (too many opportunities for invisible mistakes)

Evaluation (negative)

- ✗ More structure needed (too many opportunities for invisible mistakes)
- ✗ Limits of scale?

What We Learned

- Dumb tools can be better than smart ones (esp. when you don't know how to build the smart ones anyway)

What We Learned

- Dumb tools can be better than smart ones (esp. when you don't know how to build the smart ones anyway), but...
- A little structure is much better than none at all

What We Learned (cont'd)

- No magic bullet:
  - developing 1 type system is harder with TinkerType than without
  - developing 20 related type systems is easier with TinkerType than without

What We Learned (cont'd)

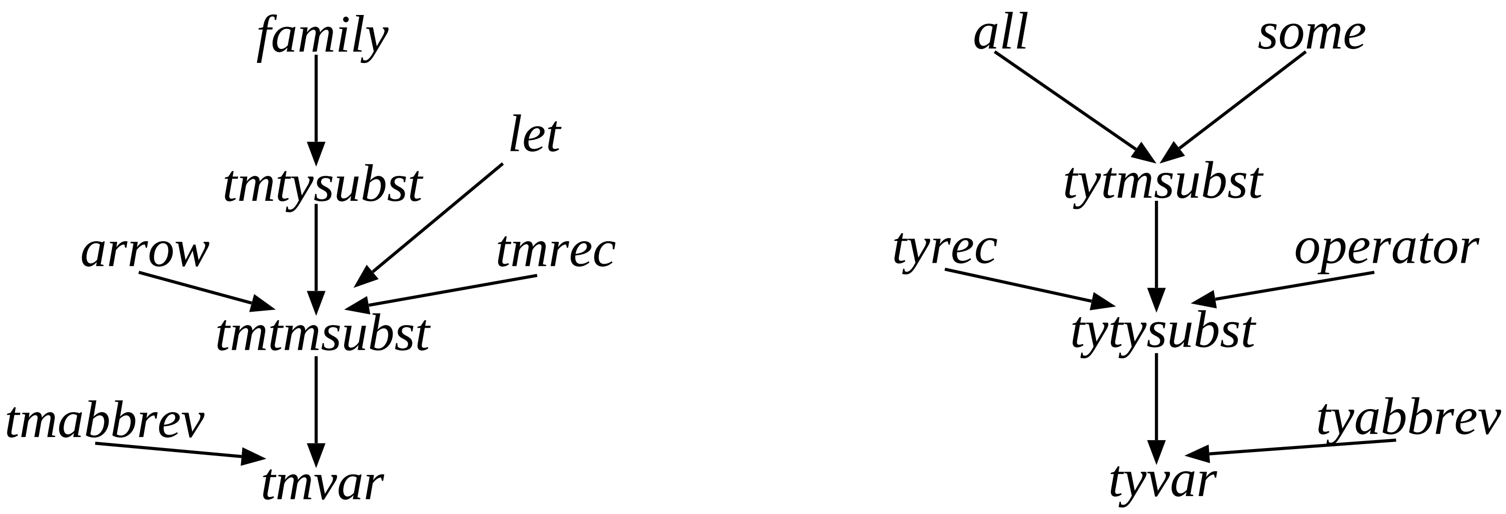
- No magic bullet:
  - developing 1 type system is harder with TinkerType than without
  - developing 20 related type systems is easier with TinkerType than without
- Inheritance and parameterization are both useful:
  - inheritance avoids the need to understand all combinations of features - supports "leafy" families of systems
  - parameterization avoids exponential blowup in orthogonal combinations of features

(cf. use of `self` in OO languages)

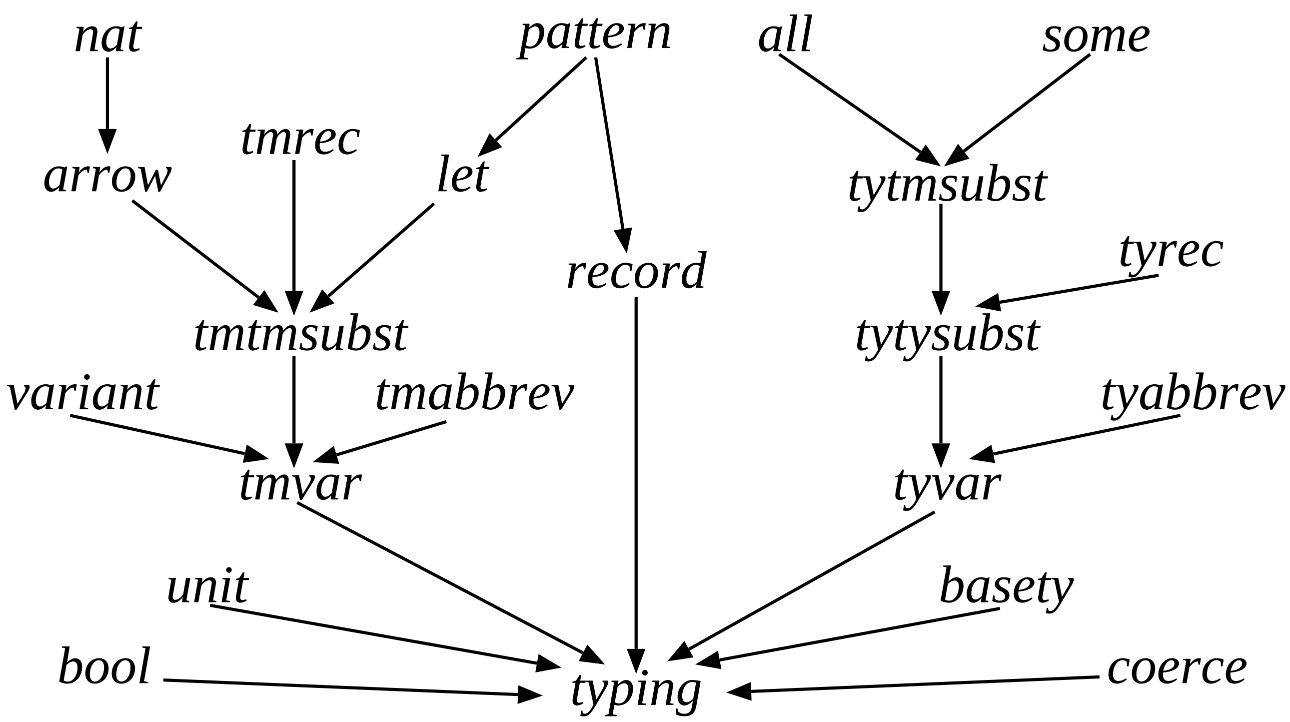
Related Work

- Abadi and Cardelli's type system fragments
- Feature-oriented programming (Prehofer)
- Aspect-oriented programming (Kiczales et al)
- Subject-oriented programming / Hyperspace (Harrison, Ossher, and Tarr)
- Monadic techniques in semantics (Moggi and gang) and action semantics (cf. Mosses talk today)

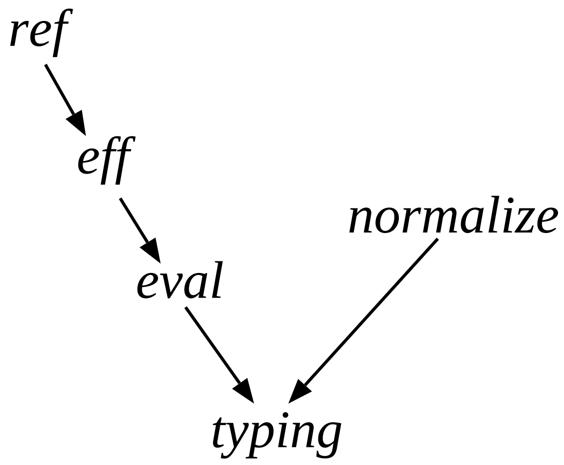
The next 700 type systems: Variable and binding features



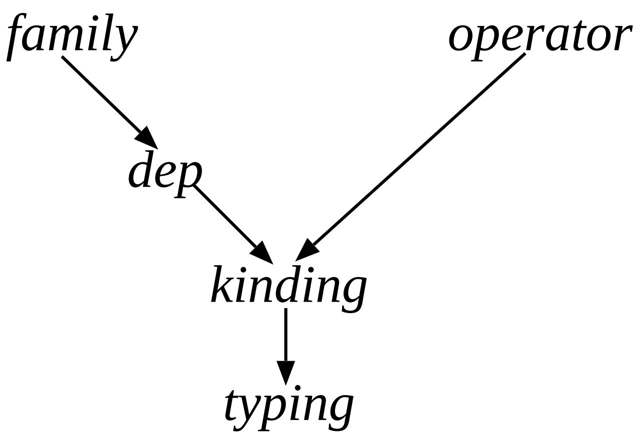
Simple features



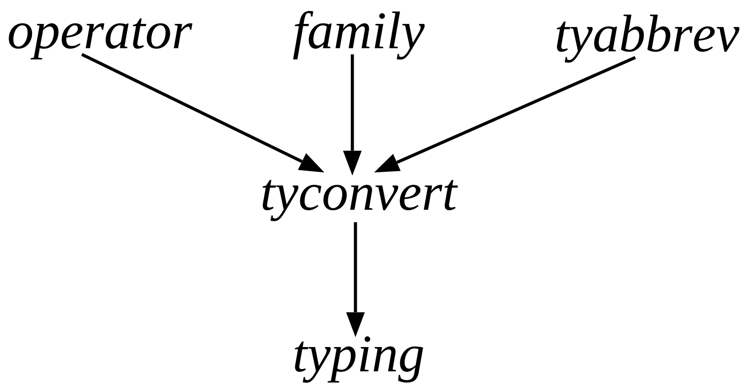
Evaluation and normalization features



Kinding features



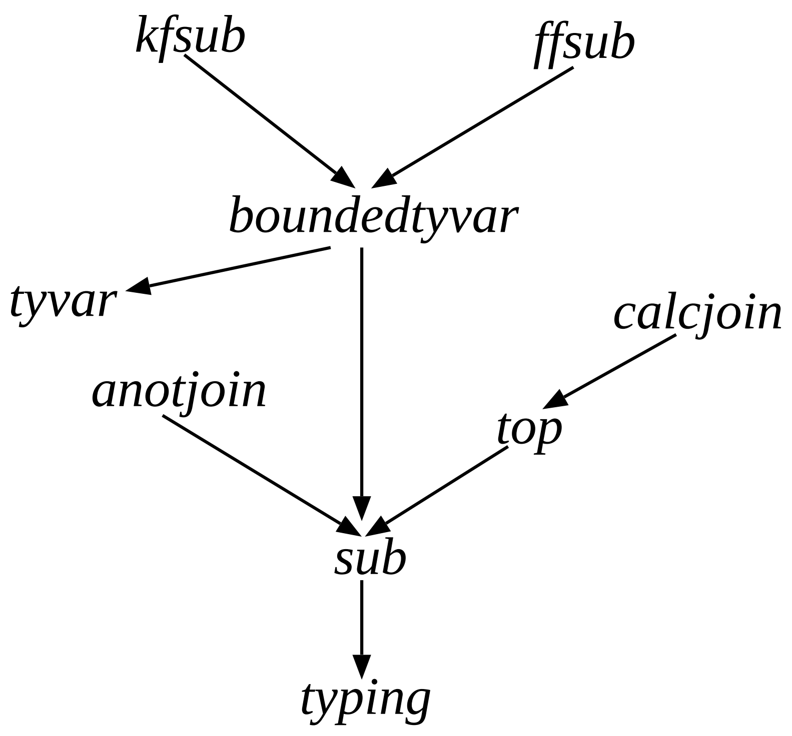
Type conversion features



The lambda-cube

[arrow]	$\lambda^{\rightarrow}$
[arrow, all]	System F
[arrow, family]	types dependent on terms
[arrow, all, operator]	System F^{ω}
[arrow, all, operator, family]	calculus of constructions

Subtyping features



[arrow, bool, calcjoin]	$\lambda^{\rightarrow}$ with subtyping, booleans and joins
[arrow, all, kfsub]	kernel $F_{<:}$
[arrow, all, ffsub]	full $F_{<:}$
[arrow, all, operator, kfsub]	kernel $F^{\omega}_{<:}$
[arrow, all, operator, ffsub]	full $F^{\omega}_{<:}$

Subtyping feature dependencies

$\mathsf{sub} \wedge (\mathsf{bool} \vee \mathsf{variant}) \Rightarrow \mathsf{calcjoin} \oplus \mathsf{anotjoin}$

$\neg(\mathsf{Kfsub} \wedge \mathsf{ffsub})$

$\neg(\mathsf{ffsub} \wedge \mathsf{calcjoin})$