On the Wasm Type System

Andreas Rossberg

Wasm CG 2023/02/28

Explicit Types

Intentional design decision in original Wasm, following extensive discussion

No overloading: {**i32**,**i64**,**f32**,**f64**}**.add** instead of just **add**

Explicit block types: **block** *blocktype instr** **end**

Motivation two-fold:

1. indicate different operational behaviour
2. make validation easy

Subtyping

Recent proposals added subtying: more power, more responsibility

Turned out ease of validation was lost in two (pre-existing) cases:

- **select**: would require computing least upper bound of input types
- **br_table**: would require computing greatest lower bound of output types

Fixed in Wasm 2.0+

- **select** by requiring type annotation (except for legacy cases)
- **br_table** by introducing bottom type and relaxing typing rule

“Modern” Wasm

Richer types and subtyping: **ref** $\$t$ <: **ref** $\$u$ and **ref** $\$t$ <: **ref null** $\$t$

Instructions that work over many reference types: **call_ref**, **struct.get**, …

Looong discussion in GC subgroup about type annotations on these instructions

Ultimately, we decided to require type annotations

- preserve the spirit of explicit types
- conservative, safer choice; avoids some complexity
- affects code size, but that eventually requires a more systematic solution

“Modern” Wasm (2)

Changes to design of **br_on_cast** instruction had the discussion arise again

It wasn’t clear what *exactly* we agreed upon last time!

…different folks had different motivation and interpretation of “explicit” types

No specification or documentation of the intended property

Risk of ad-hoc decision that negates intentions of last ad-hoc decision

Wanted: a condition that is general and implementation-independent

Principal Types

Principal Types Property

Every valid program phrase E has a most precise type T
that is a subtype of all types that it can legally have.

(T only depends on the declarations in scope, not on the surrounding program text!)

Type systems with subtyping usually want this property

- Generalises more basic “Unique Types” property of systems without subtyping
- Ensures that type checking is composable and does not require back tracking

Well-established for expression-based languages (*program phrase* = expression)

For generic operators, principal types are typically only expressible with type variables

Principal Types in Wasm

Relevant program phrases are instruction (sequences) as the unit of composition

Refine design goal of "explicit type" to "principal type" for every instruction

…that is determined only by the instruction itself and its immediates

…consequently, if immediates are not enough to narrow down the principal type of an instruction, then it needs a suitable type immediate (a.k.a. annotation)

For example, **struct.get** $\$t\ i$: (**ref null** $\$t$) → **i32** is principal, thanks to $\$t$

We can not just state this, we can formally prove this property (easily)

Principal Types for Wasm

Every valid instruction sequence instr has a (unique) type $t_1^* \rightarrow t_2^*$*
(possibly containing some place-holder type variables)
that is a subtype of all types $t'_1{}^* \rightarrow t'_2{}^*$ it can legally have
(after substituting type variables with specific types).

May involve type variables for value, heap, and stack types:

drop : $t \rightarrow \varepsilon$	for all value types t
ref.as_non_null : (ref null *ht*) → (ref *ht*)	for all heap types *ht*
unreachable : $t_1^* \rightarrow t_2^*$	for all stack types t_1^* and t_2^*

(Special case for legacy **select**, requires a variable ranging over number and vector types only)

Type variables are unconstrained, i.e, denote arbitrary choice

(Closed) Principal Forward Types

If a (closed) input type $t_1{}^*$ is given,
and instruction sequence $instr^*$ has a type $t_1{}^* \rightarrow \ldots$,
then it also has a (closed) type $t_1{}^* \rightarrow t_2{}^*$ (or $t_1{}^* \rightarrow \bot^*\ t_2{}^*$)
that is a subtype of all types $t_1{}^* \rightarrow t'_2{}^*$ it can legally have.

That's what enables efficient forward validation without type variables

- operates on *complete* instruction sequences,
  hence always starts from an empty (i.e., known) input type

Note: most type annotations are not needed for this to hold

struct.get i : (ref null $\$t$) → i32 would be forward principal, but not principal

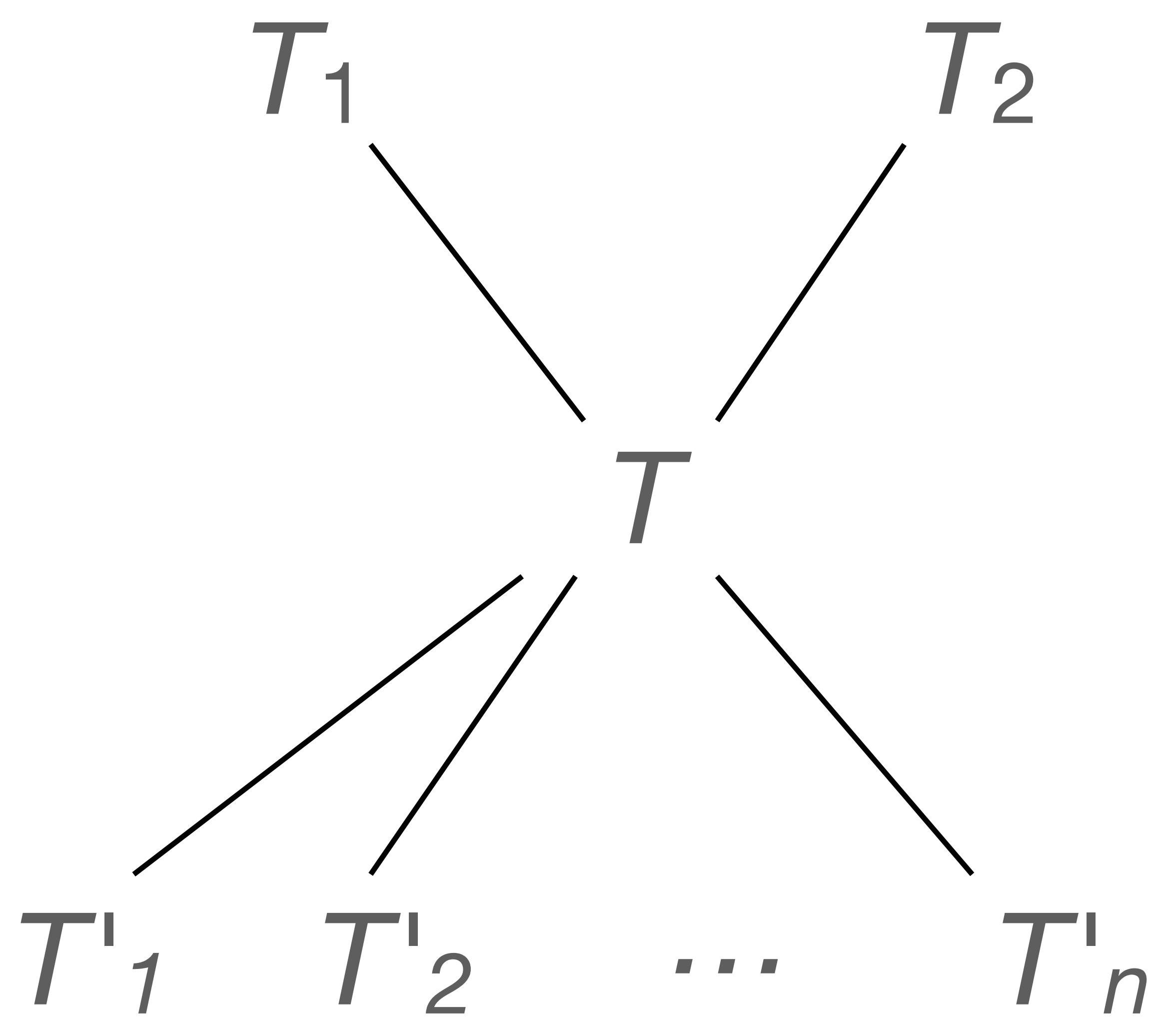
br_on_cast $\$l$ rt : $rt_1 \rightarrow rt'_1$ likewise

br_on_cast $\$l$ rt_1 rt_2 : $rt_1 \rightarrow rt'_1$ principal

Other Properties

Existence of glbs

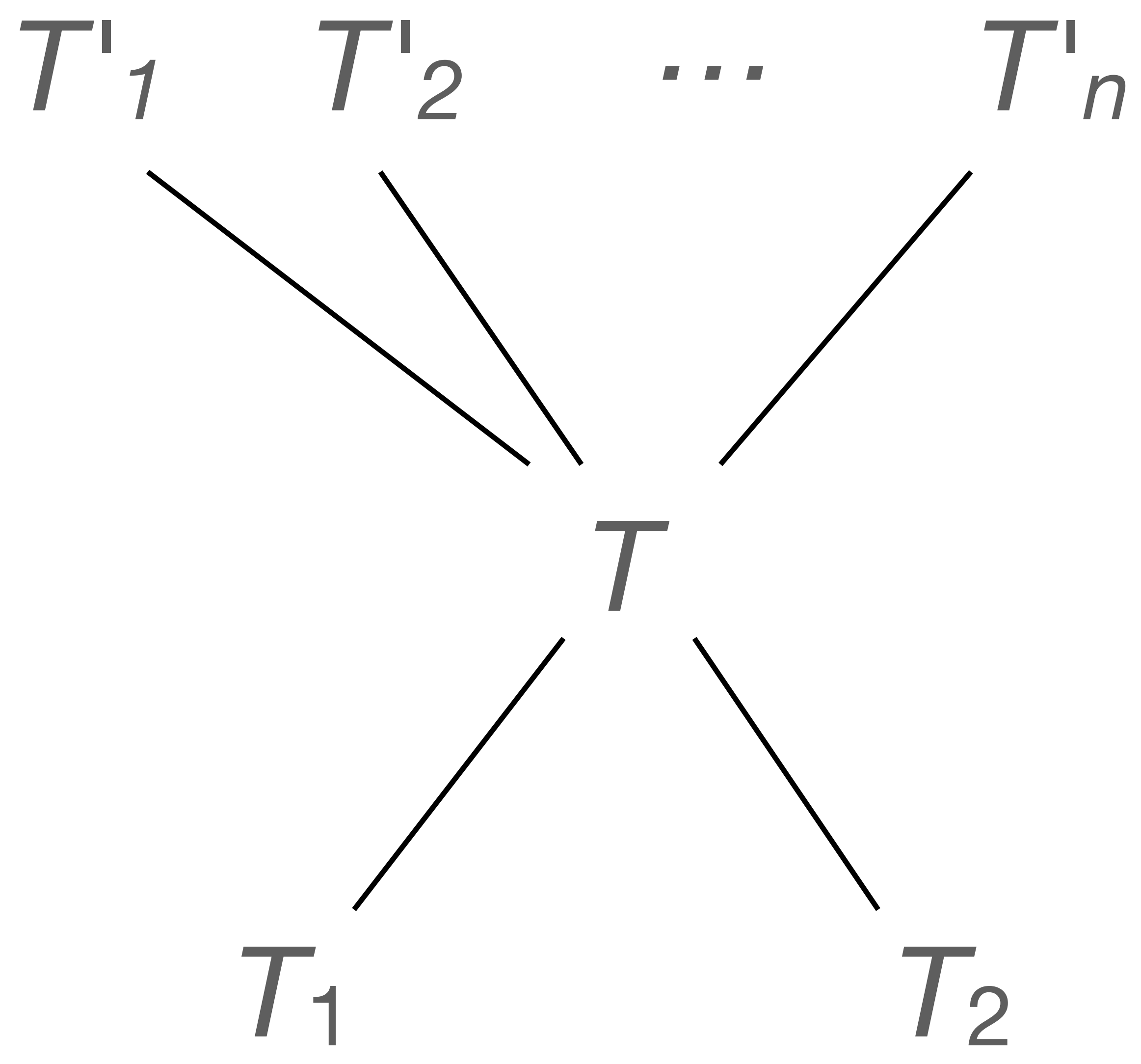
Every pair T_1 and T_2 of types
has a common subtype T
that is a supertype of all other common subtypes.
(T may be bottom.)



- In other words, subtyping forms a semi-lattice.
  Relevant for both tooling and for Principal Types.

Conditional Existence of lubs

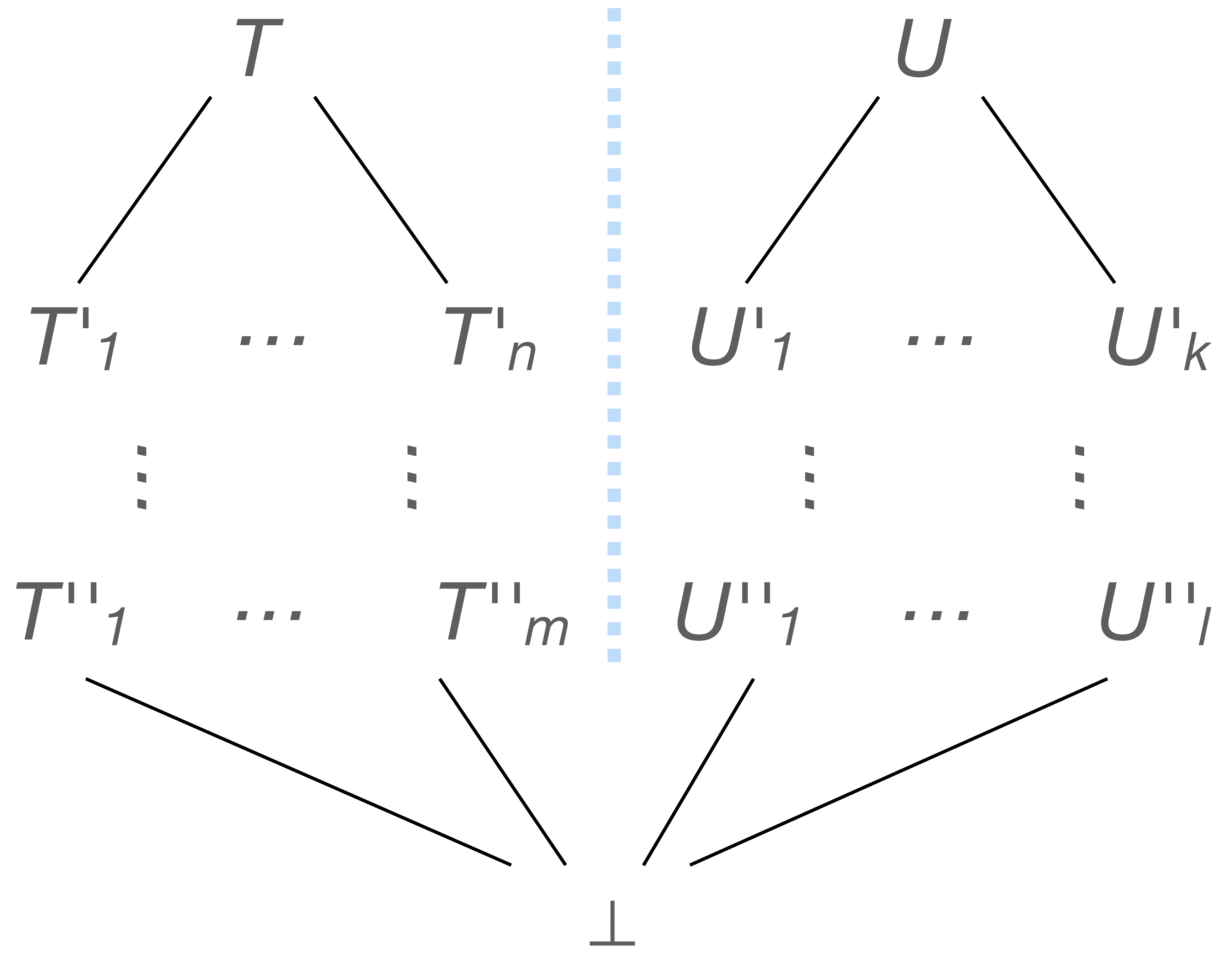
Every pair T_1 and T_2 of types
that has a common supertype at all
has a common supertype T
that is a subtype of all other common supertypes.



- In other words, if we were to add a provisional top type, we would have a proper lattice.

Disjoint Type Hierarchies

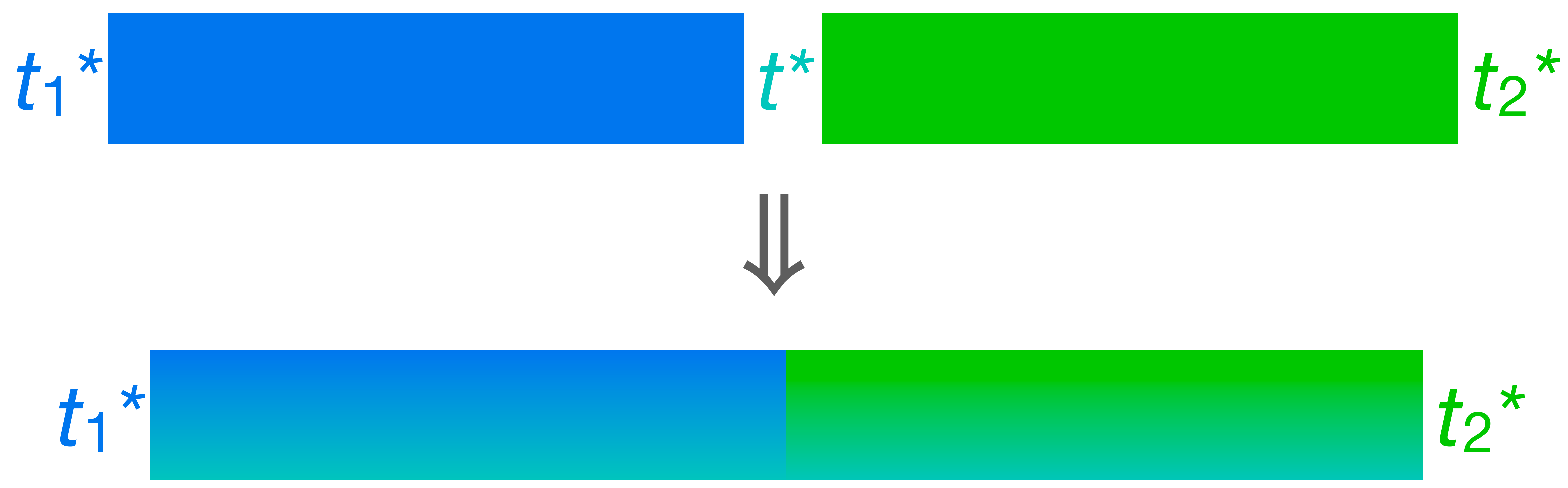
The glb of two types is bottom
if and only if
they have no common supertype.



- Values from separate type hierarchies can never flow to the same place and can hence safely be implemented with incompatible representations.

Composition

If two instruction sequences $instr_1{}^*$ and $instr_2{}^*$ are valid with types $t_1{}^* \rightarrow t^*$ and $t'^* \rightarrow t_2{}^*$, respectively, where $t^* <: t'^*$, then the combined sequence $instr^* = instr_1{}^*\ instr_2{}^*$ is valid with type $t_1{}^* \rightarrow t_2{}^*$.

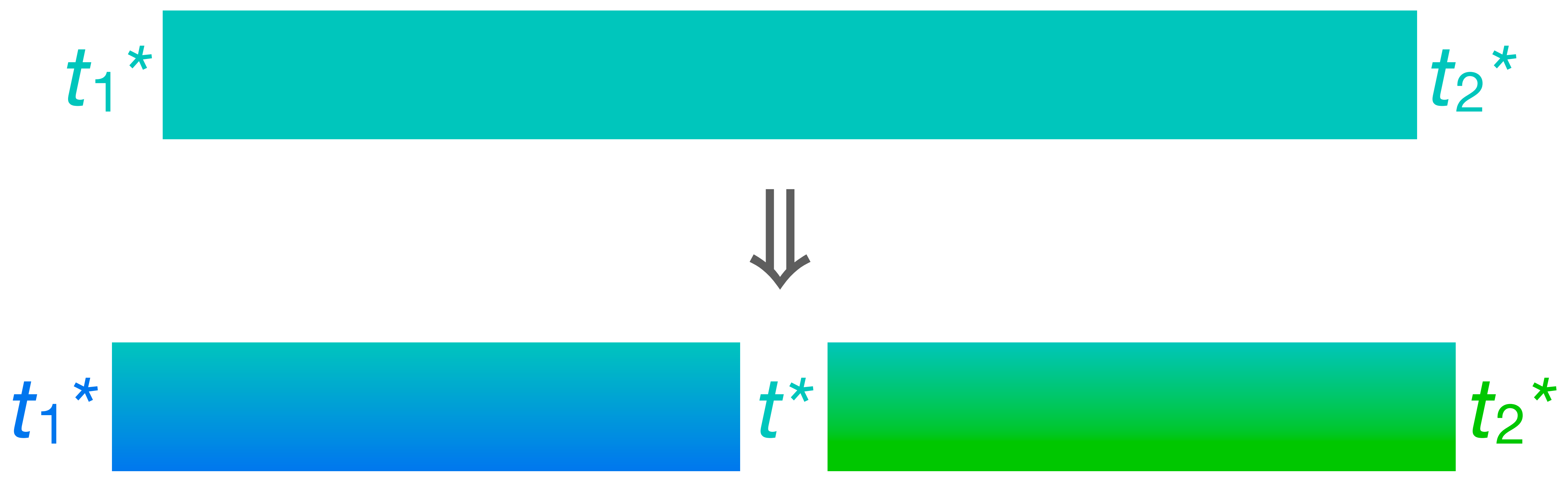


- This is straightforward.

Decomposition

If an instruction sequences $instr^*$ is valid with type $t_1{}^* \to t_2{}^*$, then any split into two instruction sequences $instr_1{}^*\ instr_2{}^* = instr^*$, is valid with some types $t_1{}^* \to t^*$ and $t^* \to t_2{}^*$.

- Depends on typing of unreachable code.



Summary

Summary of Wasm Type System Properties

Essential:

- Soundness
- (Closed) Principal Forward Types
- Conditional Existence of lubs
- Disjoint Type Hierarchies
- Composition

Desirable:

- Principal Types
- Existence of glbs
- Decomposition

Now all collected in Appendix A.3 of Typed References Proposal spec draft

Outtakes

Explicit Types vs Subtyping

What does “explicit types” mean in the presence of subtyping?

Via subsumption, instructions can naturally have many different types

struct.get $\$t\ i$: [(**ref** **null**$^?$ $\$t'$)] → [**i32**]

for any $\$t' <: \$t$
with or without **null**

Example: br_on_cast

br_on_cast $\$l\ rt\ :\ rt_1 \rightarrow rt_2$

iff rt_l <: type($\$l$)

and rt <: trt and rt_1 <: trt for some reference type trt

and $rt.\mathrm{heaptype} = rt_l.\mathrm{heaptype} \wedge rt_1.\mathrm{heaptype} = rt_2.\mathrm{heaptype}$

and $rt.\mathrm{null} = rt_l.\mathrm{null} \neq rt_2.\mathrm{null} \vee rt_1.\mathrm{null} = rt_2.\mathrm{null} = rt_l.\mathrm{null} = \varepsilon$

Requires distinguishing 6 different cases and checking 2 types for null in validator

Example: br_on_cast

br_on_cast $\$l\ rt_1\ rt_2\ :\ rt_1 \rightarrow rt'_1$

iff rt_2 <: type($\$l$)

and rt_2 <: rt_1

and rt'_1.heaptype = rt_1.heaptype

and rt'_1.null = rt_1.null $\wedge \neg$ rt_2.null