Sums of Uncertainty: Refinements Go Gradual

Long version of paper to appear at POPL 2017, including supplementary material

Khurram A. Jafery Jana Dunfield
University of British Columbia
Vancouver, Canada
kjafery@cs.ubc.ca jd169@queensu.ca

Abstract

A long-standing shortcoming of statically typed functional languages is that type checking does not rule out pattern-matching failures (run-time match exceptions). Refinement types distinguish different values of datatypes; if a program annotated with refinements passes type checking, pattern-matching failures become impossible. Unfortunately, refinement is a monolithic property of a type, exacerbating the difficulty of adding refinement types to non-trivial programs.

Gradual typing has explored how to incrementally move between static typing and dynamic typing. We develop a type system of gradual sums that combines refinement with imprecision. Then, we develop a bidirectional version of the type system, which rules out excessive imprecision, and give a type-directed translation to a target language with explicit casts. We prove that the static sublanguage cannot have match failures, that a well-typed program remains well-typed if its type annotations are made less precise, and that making annotations less precise causes target programs to fail later. Several of these results correspond to criteria for gradual typing given by Siek et al. (2015).

Categories and Subject Descriptors F.3.3 [Mathematical Logic and Formal Languages]: Studies of Program Constructs—Type structure

Keywords gradual typing, refinement types

1. Introduction

A central feature of statically typed functional languages is pattern matching over user-defined datatypes that combine several fundamental constructs: sum types (for example, an element of a bool datatype can be either True or False), recursive types (such as lists), and polymorphic types. The aspect of ML datatypes that corresponds to sum types is the focus of this paper.

Static typing is said to catch run-time errors—at least, errors that would manifest in a dynamically typed language as tag check failures, such as subtracting a string from a number. Using the venerable encoding of dynamic typing as injections into a datatype Dynamic (Abadi et al. [1991]), these tag check failures become errors raised in the “fall-through” arm of a case expression over Dynamic. The impossibility of such errors is a convincing argument in favour of static typing.

Yet Standard ML programmers frequently write code that is essentially the same as the scorned operations on Dynamic—and that has the same unfortunate risk of run-time errors. The definition of SML (Milner et al. [1997]) requires compilers to accept nonexhaustive case expressions, which do not cover all the possible instances of the datatype. A nonexhaustive case expression is isomorphic to an implicit tag check over Dynamic: the non-error case is the only one written out explicitly, while an error case is inserted by the sneaky compiler.

In fairness, the definition encourages compilers to warn about nonexhaustive case expressions. But this only causes programmers to write their own “raise Match” arms, even when the fall-through case is impossible because of an invariant known by the programmer. This leads to verbose code. In response, Freeman and Pfenning (1999) developed datasort refinements that can encode many invariants about datatypes, allowing compilers to accept “nonexhaustive” case expressions when they are known to cover all possible cases. For case analyses of refined types, the nonexhaustiveness warning becomes a nonexhaustiveness error, which the programmer should solve by declaring and using refinements of the datatype.

Unfortunately, this approach is all-or-nothing: either a type is refined and the compiler rejects a nonexhaustive match over it, or the type is not refined and the compiler issues a noncommitental warning. In practice, programmers may want to migrate code written with unrefined types to code that uses refined types; doing this in a single pass over a nontrivial program is extremely difficult. Instead, programmers should be able to add type annotations gradually. This was essentially the motivation for gradual typing (Siek and Taha [2006]), except that, where they contemplated migration from dynamically typed code to statically typed code, we are interested in migration from code that is statically typed (modulo nonexhaustiveness) to code that is more statically typed.

Gradual typing is about the possibility of uncertainty: in some cases, one knows exactly what type one has; in other cases, one does not even know whether something is an integer. In this paper, we always know whether something is an integer (or a function, etc.); uncertainty is possible, but only about sum types. This is like the uncertainty of SML datatypes, with one key difference: we allow SML-style uncertainty and refinement-style certainty.

As an example, consider a red-black tree library that passes the SML type checker, but does not use refinement types. Datasort refinements can express the colour invariant, which says that every red node’s children must be black. By reasoning about how the library functions should work, a programmer can add annotations that say when the colour invariant should hold, which the refinement type checker will verify. With gradual refinements, this reasoning can be done gradually and in tandem with testing. In fact, the programmer could start by annotating a single function $r$. If all test cases use $r$ in accordance with its refinement type annotation, the programmer gains confidence that the annotation is correct; if any tests violate the annotation, then either the annotation is wrong,
Contributions. We make the following contributions:

- We define a type assignment system of gradual sums that includes both static refinement sums and dynamic sums. Programs, and even individual types, can be partly static and partly dynamic. However, this system does not readily yield an algorithm, and it allows typing derivations that are gratuitously dynamic (more dynamic than indicated by the programmer’s type annotations), which give rise to gratuitous run-time errors.

- We define a bidirectional type system that is easy to implement and suppresses gratuitous dynamicism, and prove that it corresponds to the type assignment system. We also prove that a well-typed program remains well-typed if its type annotations are made less precise (more dynamic).

- We define a type-directed translation to a target language with explicit casts. We prove that, given one program with two sets of type annotations (one more precise than the other), the more precisely typed one “fails earlier”: either they produce the same result, or they both fail, or the more precisely typed program fails earlier. (For technical reasons, part of this result uses a slightly different version of the translation.)

- We define static and dynamic fragments of the source type system. The static fragment is related to classic datasort refinement systems; the dynamic fragment is related to Standard ML. The static fragment is related to Standard ML. Moreover, they can be freely combined. We prove that translating a program in the static fragment yields a program that cannot raise Raise.

The three sum types +, +1, and +2 are essentially a datasort refinement system. Following datasort refinement systems, we can express that a value is either +1 or +2. We also make + for a subtype of +: the only elimination form permitted for + is a two-armed case, which is always safe. But +2 must not be a subtype of +1 and +2, because +2 contains both left and right injections; through subsumption, we could use a one-armed case on the left injection inj1 to eliminate a value of type +2, which would fail at run time.

This yields the following subtype relation:

\[
\begin{align*}
A_1 + A_2 & \quad A_1 +1 A_2 \quad \text{steps to } A_1 +2 A_2 \\
A_1 +1 A_2 & \quad A_1 +2 A_2
\end{align*}
\]

For brevity, we can omit A1 and A2 from the diagram.

Comparison to datasort refinements. Our type \( A_1 + A_2 \) corresponds to the top datasort of a datatype—the datatype that contains all the values of that datatype. A case expression on + must provide two arms, one for each injection.

Our type \( A_1 +1 A_2 \) corresponds to a datatype that includes exactly the values of the form \( c_1(v_1) \) where \( v_1 : A_1 \); similarly, \( A_1 +2 A_2 \) corresponds to a datatype whose values are \( c_2(v_2) \) where \( v_2 : A_2 \).

In contrast, our type \( A_1 +^2 A_2 \) corresponds to the unrefined datatype. In datasort refinement systems, unrefined datatypes are part of the unrefined type system; the top datasort for a datatype contains the same values as the unrefined datatype, and is often noted in exactly the same way—but the unrefined datatype is not usable as a datatype. In contrast, both + and +2 are types in our system. Moreover, they can be freely combined.

2. Overview

We define a type system that has one of the essential capabilities of datasort refinements: the types can express the knowledge that a value is a particular alternative of a datatype; for example, that a value is not simply a list—either Nil or Cons(…)—but specifically Cons(…). We represent this knowledge through sum types, not through the usual form of datasort refinements, but that is not the important difference.

- Like conventional datatype systems and datasort refinement systems, we can express that a value is either inj1 \( e_1 \) where \( e_1 \) has type \( A_1 \) or inj2 \( e_2 \) where \( e_2 \) has type \( A_2 \). Like datasort refinement systems, we only allow an exhaustive (two-armed) case expression over such a type: if we don’t know which injection it is, the programmer must handle both cases. This is a standard sum type \( A_1 + A_2 \).

- Like datasort refinement systems, we can express that a value must be a particular injection. We use a subscript sum \( A_1 +_k A_2 \) for the type of the \( k \)th injection into \( A_1 + A_2 \). For example, \( \text{inj}_1 \) True has type Int +_2 Bool, but \( \text{inj}_1 \) 5 has type Int +_1 Bool. Also like datasort refinement systems, we allow case expressions over such types to have just one arm, because we know which injection we have; there is no need to handle an impossible case.

- Like conventional datatype systems, but unlike datasort refinement systems, we can also express that we don’t know which injection we have; \textit{but want to allow nonexhaustive matches}: the \textit{dynamic sum} \( A_1 +^2 A_2 \) can be deconstructed by a one-armed case expression. If, at run time, the specified arm does not match the scrutinee, it is a run-time error.

Figure 1 depicts some of the results: source programs \( e \) are translated to target terms \( M \), which step to \( M' \); preserving typing; source programs \( e^S \) with only static types are translated to target terms with no match failures. For space reasons, lemmas, proofs, and a few definitions can be found in the supplementary material.

2.1 Developing Typing and Subtyping

Verificationists and pragmatists. In the verificationist approach to type theory, followed by Gentzen (1934) and Martin-Löf (1971), introduction forms are taken as the definition of a type; for example, a boolean type is defined by its constructors True and False. The elimination forms are secondary. In the pragmatist approach considered by Dummett (1991) and Zeilberger (2009), elimination forms are taken as the definition, and the introduction forms are
secondary. For example, a boolean type is defined primarily by its elimination form (say, an if-then-else expression).

In our setting, neither strict verificationism nor strict pragmatism seems adequate. Verificationism serves refinements well: the introduction rules directly express the intuition that refinements identify subsets of values. But introduction rules alone cannot distinguish \( A_1 + A_2 \) and \( A_1 +^2 A_2 \), because they have identical sets of inhabiting values (namely, all \( \text{inj}_1 \), \( v_1 \) and \( \text{inj}_2 \), \( v_2 \) such that \( v_1 : A_1 \) and \( v_2 : A_2 \)). The difference must lie in the elimination forms: only a two-armed case can eliminate \(+\), while \( +^2 \) can be eliminated by a two-armed case or a one-armed case (since the point is to allow nonexclusive matches). To start from a better-understood foundation, we begin with the introduction rules.

Designing a type system can require trading off simplicity in one set of rules for complexity in another. We choose to minimize the number of typing rules, even though it leads to more complicated subtyping.

**Introduction rules.** Sum types need introduction forms. Since \(+\) should contain only left injections, and \(+^2\) should contain only right injections, we could have a rule

\[
\Gamma \vdash e : A_k \\
\Gamma \vdash (\text{inj}_k e) : (A_1 + +_2 A_2) +^k \text{Intro}
\]

(This rule is really two rules, one for \( \text{inj}_1 \) with a premise \( \Gamma \vdash e : A_1 \) and one for \( \text{inj}_2 \) with a premise \( \Gamma \vdash e : A_2 \).)

Combined with subsumption, this rule gives the desired inhabitants to \(+\), that is, both left and right injections. However, it does not add any inhabitants to \( +^2 \), so we could add another rule:

\[
\Gamma \vdash e : A_k \\
\Gamma \vdash (\text{inj}_k e) : (A_1 + +^2 A_2) +^k \text{Intro}
\]

This goes against our goal of minimizing the number of typing rules: now there are two rules that type \( \text{inj}_k e \) directly, that is, without using subsumption. The types \(+^k\) (given by \( +^k \text{Intro} \)) and \( +^2\) (given by \(+^2 \text{Intro} \)) are not in a subtyping relation with each other—neither is a subtype of the other. Hence, neither rule encompasses the other, and both are required.

We can avoid this nondeterminism by adding more sum types. By placing the additional sum types at the bottom of the subtyping relation, we can write a single introduction rule that will (through subsumption) populate all of our types with the desired injections.

\[
\begin{array}{c}
\text{+} \\
\text{+} \\
\text{+} \\
\text{+^2} \\
\text{+^2} \\
\end{array}
\]

Now, we need only one introduction rule:

\[
\Gamma \vdash e : A_k \\
\Gamma \vdash (\text{inj}_k e) : (A_1 + +^2 A_2) +^k \text{Intro}
\]

We can think of \( +^2 \) and \( +^2 \) as “innate” types: when an injection \( \text{inj}_k \) is created, it has type \( +^k \). Through subtyping, we can interpret \( +^k \) as \( +k \), or as the dynamic sum \( +^k \).

**Elimination rules.** To design the elimination rules, it is helpful to annotate the subtyping diagram with the elimination forms that each type should allow. We write \( L \) for a one-armed case expression on the left injection \( (\text{inj}_1) \), \( R \) for a one-armed case on the right injection \( (\text{inj}_2) \), and \( B \) for a two-armed case.

\[
\begin{array}{c}
L, R, B \\
\text{+} \\
\text{+} \\
\text{+} \\
\text{+} \\
\text{+} \\
\end{array}
\]

According to this diagram, all types support a two-armed case expression \( B \). The types \(+^1\) and \(+^1\) are inhabited only by \( \text{inj}_1 \), so they support the left one-armed case \( L \); similarly, \(+^2\) and \(+^2\) support the right one-armed case \( R \). However, \(+^1\) and \(+^2\) are subtypes of \( +^2 \), so by subsumption they also support the “wrong” one-armed cases. The dynamic sum \( +^2 \) supports all three eliminations, with the risk of failing at run time.

Handling the two-armed case expression is straightforward: all the sum types support that elimination form, and all the sum types are subtypes of \(+\), so we can write a single rule that types the scrutinee with \( + \). Given \( e : (A_1 + A_2) \) where \( \phi \) is any of our sum types, subsumption can be used to derive \( e : (A_1 + A_2) \).

\[
\begin{array}{c}
\Gamma \vdash e : (A_1 + A_2) \\
\Gamma, x_1 : A_1 \vdash e_1 : B \\
\Gamma, x_2 : A_2 \vdash e_2 : B \\
\Gamma \vdash \text{Elim}(e, \text{inj}_1 x_1 e_1, \text{inj}_2 x_2 e_2) : B
\end{array}
\]

One-armed case expressions are more troublesome. Consider a left one-armed case, which matches only values of the form \( \text{inj}_1 \). Any subtype of \( +^1 \) will work, so we can write a rule that handles \(+^1\) and \(+^1\) (and symmetrically, \(+^2\) and \(+^2\)). However, \( +^2 \) should support a left one-armed case, but \( +^2 \) is not a subtype of \( +^1 \), leading us to a second rule that handles \( +^2 \).

Since \( +^2 \) supports one-armed cases, it violates a type-theoretic principle: the introduction and elimination rules of a logical connective should be in harmony—that is, they should be locally sound (Dummett 1991) and locally complete (Pfenning and Davies 2001). Local soundness holds when the elimination rules are not more powerful than the introduction rules. Consider some standard rules for pairs:

\[
\begin{array}{c}
\Gamma \vdash e_1 : A_1 \\
\Gamma \vdash e_2 : A_2 \\
\Gamma \vdash e : (A_1 \times A_2) \\
\Gamma \vdash \text{proj}_k e : A_k
\end{array}
\]

These rules are locally sound: given something of type \( A_1 \times A_2 \), projection can only extract things of type \( A_1 \) and \( A_2 \).

Dually, local completeness says that the elimination rules can extract all the information used in the introduction rules. (For a concise explanation of harmony, see Pfenning (2009).)

When the Curry–Howard correspondence holds, a type is inhabited iff the corresponding proposition is provable. Consider the following derivation (eliding empty contexts):

\[
\begin{array}{c}
e : A_1 \\
(\text{inj}_1 e) : (A_1 +^1 +^2 A_2) \leq (A_1 +^2 A_2) \\
(\text{inj}_1 e) : (A_1 +^2 A_2) \\
\Gamma \vdash \text{case}(\text{inj}_1 e, x) : A_2
\end{array}
\]

By constructing \( \text{inj}_1 e \), we have shown that \( A_1 \) is inhabited. By subsumption, \( \text{inj}_2 e \) has type \( A_1 +^2 A_2 \). An elimination rule for \( +^2 \) must permit a one-armed case on the second injection, ostensibly having type \( A_2 \). Simply returning \( x \) as the result of the \( \text{case} \) should show that the proposition corresponding to \( A_2 \) is provable. But we never constructed something of type \( A_2 \), so \( +^2 \) does not satisfy local soundness.
As we did for the introduction forms, a single elimination rule can suffice: we just need more sum types. For the introduction forms, we added types at the bottom of the subtyping relation. Since eliminations should behave dually, we will add types at (or, at least, near) the top of the subtyping relation.

The types \( +_1^2 \) and \( +_2^1 \) support exactly the same eliminations as the subscript sums \( +_1 \) and \( +_2 \), but unlike the subscript sums, they are supertypes of the dynamic sum \( +^2 \).

Then the single elimination rule for one-armed cases is

\[
\Gamma \vdash e : (A_1 +^*_k A_2) \quad \Gamma, x : A_k \vdash e_k : B \\
\Gamma \vdash \text{case}(e, \text{inj}_k x, e_k) : B \\
\begin{array}{c}
\text{Elim}
\end{array}
\]

We could simplify the diagram slightly by removing the edge from \( +^2 \) to \( + \), since we now have an alternate routing via the \( +_2^1 \) types.

**The high-water mark.** Have we added enough sum types? We believe so. First, the additional types (beyond \( +_1, +_2, +^1, +^2 \)) are motivated by limiting the number of typing rules. Second, there seem to be no other types that could be useful. Consider the following table:

<table>
<thead>
<tr>
<th>inhabitants</th>
<th>elimination forms supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{inj}_1 )</td>
<td>B only ( + )</td>
</tr>
<tr>
<td>( \text{inj}_2 )</td>
<td>note (a) ( +_1 )</td>
</tr>
<tr>
<td>( \text{inj}_1, \text{inj}_2 )</td>
<td>( +_1 )</td>
</tr>
</tbody>
</table>

In the spaces marked “note (a)”, such a type would pointlessly restrict the possible elimination forms: the top left space would be a type that could only be eliminated by a two-armed case (“B only”), but was inhabited only by left injections \( \text{inj}_1 \).

In the spaces marked “note (b)”, such a type would allow one-armed cases that always fail: a left one-armed case L on \( \text{inj}_2 \), or a right one-armed case R on \( \text{inj}_1 \). We provide \( +^2 \) to give programmers the freedom to use one-armed cases that may fail; it seems pointless to give them one-armed cases that are guaranteed to fail.

If anything, we may have more sum types than we want in practice: having fewer typing rules is good, but showing \( +_1^2 \) or \( +_2^1 \) in a compiler error message seems unhelpful.

### 2.2 Developing Precision

Our ultimate goal is a language in which precisely typed code and imprecisely typed code can coexist. In precisely typed code, the impossibility of match failures is a consequence of typing. In imprecisely typed code, bugs may lead to match failures, but imprecisely typed code can be correct: a one-armed case expression may be exhaustive in practice, thanks to some invariant not expressed through the type system.

The approach to typing and subtyping, developed above, already permits some forms of coexistence. For example, if a function \( f \) expects a sum type \( + \) and we have some \( x \) of type \( +^2 \), we can pass \( x \) to \( f \). In the derivation below, \( \Gamma = f : (A_1 + A_2) \rightarrow B, x : (A_1 +^2 A_2) \).

\[
\begin{array}{c}
\Gamma \vdash x : A_1 +^2 A_2 \\
\Gamma \vdash x_1 : A_1 + A_2 \\
\Gamma \vdash x_2 : A_2 +^2 A_2 \\
\Gamma \vdash \text{case}(x, \text{inj}_1 x_1, \text{inj}_2 x_2) : B
\end{array}
\]

What about the reverse situation? Suppose a function \( g \) from the imprecisely typed part of the program expects \( +^2 \), and we want to pass something of type \( + \). This is possible, but annoying: we have to use a two-armed case to decompose the sum, and immediately rebuild it at type \( +^2 \). Here, \( \Gamma = g : (A_1 +^2 A_2) \rightarrow B, y : (A_1 + A_2) \).

\[
\begin{array}{c}
\Gamma \vdash f \ : \ (A_1 + A_2) \rightarrow B \\
\Gamma \vdash x : A_1 + A_2 \\
\Gamma \vdash y : A_1 + A_2 \\
\Gamma \vdash \text{case}(y, \text{inj}_1 x_1, \text{inj}_1 x_1, \text{inj}_2 x_2, \text{inj}_2 x_2) : B
\end{array}
\]

To support directly calling imprecise code from precise code, we develop precision relations on sum constructors and types. These relations are inspired by precision relations developed in gradual typing, e.g. Siek and Vachharajani (2008) and Garcia et al. (2016), where \( ? \) (or \( * \)) is an unknown, and thus very imprecise, type.

Our static sums \( +, +_1, +_2 \) are precise in the sense that the “reach” of their information is known. If we have a closed value \( v \) of type \( A_1 + A_2 \), the type system “knows” only that \( v \) is either a left or right injection, with no further information. So the type system rejects a one-armed case on \( v \).

On the other hand, the dynamic sum \( +^2 \) is imprecise. Some programs that use \( +^2 \) will have run-time match failures, but some programs that use \( +^2 \) will not have such failures, even some that use one-armed cases. If such one-armed cases always succeed, it is because the program follows invariants that are not expressed in the types—but which may be known by the programmer.

So we would expect \( + \) to be more precise than \( +^2 \), noted \( + \subseteq +^2 \) (which can also be read “\( + \) is less imprecise than \( +^2 \)”). What about \( +_1 \) and \( +_2 \)? They should be more precise than \( +^2 \); indeed, \( +^2 \) should be more imprecise than everything else. How do \( +_1 \) and \( +_2 \) compare? It is true that \( +_1 \) has fewer inhabitants than \( + \), but precision is not subtyping. All the static sums have the same degree of certainty: they are equally certain about different propositions (being a left injection, being a right injection, or being either). Thus, we will put \( +_1, +_2 \) and \( + \) together at the bottom of the precision relation \( \leq \) (they are the least imprecise), with \( +^2 \) at the top:

\[
\begin{array}{c}
+^7 \\
+^2 \\
+_1 \\
+_2 \\
+ \\
+^2
\end{array}
\]

What properties should precision have? In gradual typing, an important property of precision is that a program should remain well-typed when type annotations are made less precise. In the limit, we should be able to replace all static sums in annotations with \( +^2 \). We call this property *varying precision*: it is part of the “gradual guarantee” of Siek et al. (2015). (Making annotations more precise does not necessarily preserve typing: for example, changing a \( +^2 \) annotation on \( \text{inj}_2 \) to \( +_1 \).)

This property reinforces the intuition that \( +^2 \) should be at the top: this is what lets us substitute \( +^2 \) for more-precise sums. Dually, the static sums should be at the bottom: replacing a sum with a static sum should not, in general, preserve typing.

With this property in mind, how precise are \( +_1^2 \) and \( +_2^1 \), which we put in to reduce the number of typing rules? It doesn’t make sense to "mix subscripts": moving between \( +_1^2 \) and \( +_2^1 \) in an annotation, or between \( +_1 +_2 \), never preserves typing. Types with 1 subscripts should stay on the left of the edge from \( + \) to \( +^2 \), and 2 subscripts should stay on the right.
Hence, we will place $\dagger_1$ and $\dagger_1^+$ left of the vertical edge (from $\dagger$ to $\dagger^+$), and $\dagger_2$ and $\dagger_2^+$ right of the vertical edge.

Moving to a less precise type should not lose inhabitants, because the lost inhabitants will become ill-typed. Suppose we put $\dagger_2$ below $\dagger_1$, making $\dagger_1$ more precise. The sum $\dagger_1$ contains both left and right injections (by the above subtyping relation, $\dagger_2 \leq \dagger_1$), meaning that $\dagger_1$ has more inhabitants than $\dagger_1$. Therefore, we should not have $\dagger_1 \sqsubseteq \dagger_1^+$.

The reverse, where $\dagger_1^+ \sqsubseteq \dagger_1$, is more plausible but would have unfortunate consequences (discussed at the end of this section). So we have no edge between $\dagger_1^+$ and $\dagger_1$.

Lifting this relation $\sqsubseteq$ on sum constructors to sum types is straightforward: if $\delta' \sqsubseteq \delta$ then $(A_1 \delta A_2') \sqsubseteq (A_1 \delta A_2)$, provided $A_1' \sqsubseteq A_1$ and $A_2 \sqsubseteq A_2$. For function types, we diverge from subtyping: precision is covariant in the domain and $\delta$ in the domain. This is consistent with precision in gradual typing, e.g. Siek and Vachharajani (2008) and García et al. (2016), and with the refinement relations of Freeman (1994, p. 31) and Davies (2003).

Can we use this relation to type the above example g y, where we want to pass a value of type $+$ to a function expecting something of $\dagger$ type? Subtyping is internalized through a subsumption rule (the rule on the left); we extend the rule to allow loss of precision: in addition to moving from $A$ to a supertype $B$, we can move from $B$ to a less-precise $B'$.

\[
\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B} \quad \text{(sub)}
\]

Imprecision is fundamentally unsound: Using $B \sqsubseteq B'$, we move from a precise type (containing, say, $+$ and $\dagger$) to an imprecise type containing $\dagger$. Above, we showed that $\dagger$ does not satisfy local soundness. The purpose of the $B \sqsubseteq B'$ premise is to allow more-precisely-typed code to interface with less-precisely-typed code. However, a type checker that lost precision wherever possible would behave like a type checker for a system that only had $\dagger$.

In addition to losing precision after subtyping, we allow gaining precision before subtyping:

\[
\frac{\Gamma \vdash e : A' \quad A \sqsubseteq A' \quad B \leq B'}{\Gamma \vdash e : B'} \quad \text{(gain+sub+loss)}
\]

Gaining precision is clearly unsound: $A \sqsubseteq A'$ allows moving from $\dagger$ to $\dagger_1$ or $\dagger_2$. While unsound, this is needed for the property of varying precision: the typing of a single part of a program can become more or less precise, independent of the typing of the rest of the program.

We compose the three premises—gaining precision $A \sqsubseteq A'$, subtyping $A \leq B$, and losing precision $B \sqsubseteq B'$—into a relation $A' \sim B'$, called directed consistency.

With this relation, allowing $\dagger_1' \subseteq \dagger_1$ would nearly erase the distinction between $\dagger_1$ and $\dagger_2$: first, $\dagger_1' \subseteq \dagger_1$; second, $\dagger_1' \leq \dagger_2$; third, $\dagger_2 \subseteq \dagger_2$. (An earlier version of our system did allow $\dagger_1' \subseteq \dagger_1$—see Appendix C)

Ideally, we should apply imprecision only when the programmer intends it. This goal motivates the bidirectional system in Section 4.

3. Source Type System

\[
i := 1 \mid 2
\]

Source sums

\[
\delta ::= + | +^+ | +^\sharp | +^1
\]

Source expressions

\[
e ::= () | x | \lambda x. e | e_1 e_2 | (e :: A) | \text{inj}_j e | \text{case}(e, \text{inj}_j x_1, e_1, \text{inj}_j x_2, e_2) \mid \text{case}(e, \text{inj}_j x, e_i)
\]

Source types

\[
A, B ::= \text{Unit} \mid A \sqsubseteq B \mid A \rightarrow B
\]

Source typing contexts

\[
\Gamma ::= \cdot | \Gamma, x : A
\]

Figure 2. Source syntax

The syntax of the source language is in Figure 2. Here, and throughout the paper, $i$ ranges over 1 and 2. The symbol $\delta$ ranges over the sum constructors: $+$ is the standard (static) sum, $\dagger_1$ and $\dagger_2$ are subscript sums denoting the $i$th injections, and $\dagger^+$ is the gradual or dynamic sum. The final sum constructors, $\dagger_1$ and $\dagger_2$, are motivated by the desire to have the smallest number of introduction and elimination rules, as described in Section 2.

Source expressions are the unit $\Omega$, variables $x$, abstraction $\lambda x. e$ and application $e_1 e_2$, sum injection $\text{inj}_j e$, annotation (or ascription) $(e :: A)$, a two-armed case that eliminates $+$, and a one-armed case that eliminates $\dagger$.

Types $A$ and $B$ are Unit, sums $A \sqsubseteq B$, and functions $A \rightarrow B$. Typing contexts $\Gamma$ are unordered sets of typings $x : A$, where the $x$ are assumed to be distinct.

3.1 Subtyping and Precision

Figure 4 gives the rules for a subsumption judgment on sum constructors, written $\delta \leq \delta'$. These rules follow the diagram in Section 3. The subtyping rule for sum types uses the subsumption judgment. As is standard, the subtyping rule for functions is contravariant in the domain ($A_1' \leq A_1$) and covariant in the codomain ($A_2' \leq A_2$).

Precision on sum constructors (top of Figure 3) corresponds to the diagram from Section 2. On function types, precision is covariant in the domain, as discussed above.

In both subtyping and precision (for types), reflexivity and transitivity are admissible rules. Including transitivity rules would be fine on paper, but hard to implement since the middle type must be guessed. (The relations on sum constructors are a small finite set, so we do include transitivity rules; for an implementation, we would take the transitive closure.)

Subtyping and precision compose to form the directed consistency relation, which has a single rule $\text{DirConsU}$ in Figure 5. The “U” in the name comes from the depiction to the right of the rule. Since precision is reflexive, $\text{DirConsU}$ includes all pairs of types that are related by subtyping.

3.2 Typing Rules

Typing rules for the source language are shown in Figure 6. The rule for variables, $\text{SVar}$ is standard. Rules $\text{SAnno}$ and $\text{SUnitIntro}$ are standard, as are the rules $\text{SIntro}$ and $\text{SElim}$ for functions. Rule $\text{SCSub}$ is a consistent subsumption rule: if $e$ has type $A'$ and $A'$ is directed consistent (Figure 5) with $A$, then $e$ has type $A$.

The rules for sums $\text{SSumIntro}$ and $\text{SSumElim}$ were developed in Section 2.

4. Bidirectional Source Typing

Motivation. The type assignment system of Section 3 includes all the sensible sum types, along with subtyping and precision. By itself, the consistent subsumption rule $\text{SCSub}$ makes type inference, and even type-checking, nontrivial: we should apply $\text{SCSub}$ only
We solve all of these problems via a bidirectional version of the system. In many settings, bidirectional typing has been chosen to overcome fundamental limitations of type inference, such as undecidability of inference for object-oriented subtyping (Pierce and Turner 1998), dependent types (Xi and Pfenning 1999), Pientka and Dunfield (2010), and first-class polymorphism (Dunfield and Krishnaswami 2013). It can also be motivated by better localization of type error messages. Our motivation is different: we want to stop the type-checker from doing certain things unless the programmer has signalled that they really want to do those things. Programmers signal their intent through type annotations, which are propagated through the bidirectional typing rules.

In Section 4.3, we show that the bidirectional system is sound and complete (under annotation) with respect to the type assignment system of Section 3.

Checking and synthesis. Bidirectional typing splits typing into two judgments. The checking judgment \( \Gamma \vdash e : A \) is read “e checks against type A”; the synthesis judgment \( \Gamma \vdash \alpha : B \) is read “e synthesizes type A”. Both judgments can be interpreted as saying that e has type A; the difference is that in checking, the type A is already known, while synthesis infers A from the available information (\( \Gamma \) and e). The type in the checking judgment “flows” from some type annotation, either directly or (usually) indirectly.

An important advantage of the bidirectional system is a kind of subformula property (Gentzen 1934; Prawitz 1965). In our case, this property says that in a derivation of \( \Gamma \vdash e \Rightarrow A \), every type synthesized or checked against is derived from types found in \( \Gamma \) and e. For \( \Gamma \vdash e \Rightarrow A \), every such type is derived from \( \Gamma, e \), and A. Consequently, dynamic sums cannot appear out of nowhere: they result only from type annotations. We exploit this property in, for example, the proof of Theorem 5.

From type assignment rules to bidirectional rules. As is often the case with bidirectional type systems, our bidirectional rules will strongly resemble our type assignment rules. In general, we construct a bidirectional rule by replacing “\( \Rightarrow \)” with “\( \vdash \)” or “\( \Rightarrow \)”. The main question is when to use checking, and when to use synthesis. Checking is more powerful than synthesis; for a premise, we generally prefer to make it a checking judgment, but a checking conclusion may increase the number of required type annotations.
For the most part, we follow the recipe of Davies and Pfenning (2004), Dunfield and Pfenning (2004): introduction rules check, and elimination rules synthesize. More precisely, the judgment that includes the relevant connective—the principal judgment—should check for an introduction rule, and synthesize for an elimination rule.

Doing this step naturally determines the directions of many other judgments. For example, in rule Syn\text{Anno} the principal judgment is the first premise $\Gamma \vdash e \equiv A$, since the type in a synthesis judgment is output, deriving this premise tells us what $A_1$ is, enabling us to make the second premise a checking judgment. The premise also tells us what $e$ is, enabling us to make the second premise a synthesis judgment. Consequently, applications $e_1 e_2$ will synthesize a type, without any local annotation, whenever the function $e_1$ synthesizes. In rule Chk\text{Intro} not following the recipe—by making the conclusion synthesize, $\Gamma \vdash \lambda x. e \equiv (A_1 \rightarrow A_2)$—means that we don’t know $A_1$, and cannot construct the context $\Gamma, x : A_1$. The (im)possibility of designating a more complicated system in which $\lambda x. e$ does synthesize, as Dunfield and Krishnaswami (2013) did for a different type system.

Rule Chk\text{SumIntro} says that $\text{inj}_1 e$ checks against $A_1 \delta A_2$, where $\delta$ is any sum above $\rightarrow^*$; that is, any sum constructor except $\rightarrow^2$ and $\rightarrow^3$. This is a checking rule for two reasons. First, it is an introduction form, so according to the recipe its principal judgment (the conclusion) should check. Second, the simplest synthesizing rule would synthesize $A_1 \rightarrow A_2$, but that is a subtype of $A_1 \rightarrow^* A_2$, introducing a possibly undesired dynamic sum.

In the (one-armed) elimination rule $\text{SSumElim}$, the principal judgment is the premise $\Gamma \vdash e_0 : A_1 \rightarrow^* A_2$. Following the recipe, the corresponding premise of Chk\text{SumIntro} synthesizes. It would be unfortunate to require it to synthesize exactly $A_1 \rightarrow^* A_2$: assuming programmers mostly write type annotations using $\rightarrow^1, \rightarrow^2, \rightarrow^3$ virtually no expressions will synthesize $\rightarrow^4$. On the other hand, checking $e_0$ against $A_1 \rightarrow^* A_2$ would be too permissive: if we have a left one-armed case $\text{case}(e_0, \text{inj}_1 x, e)$, we would accept $e_0$ of type $\rightarrow^2$, even though $\rightarrow^2$ is a right injection, guaranteeing a runtime failure. Instead, we require that $e_0$ synthesize $A_1 \delta A_2$ where $\delta \rightarrow^* \rightarrow^3$. The judgment $\delta \rightarrow^* \rightarrow^3$ is derivable when $\delta \rightarrow^* \rightarrow^1$ or $\rightarrow^3$.

For consistency with our two-armed elimination rule $\text{ChkSumElim2}$ has a similar structure (with an additional premise for the second arm) and also uses the $\Rightarrow$ judgment; however, $\Rightarrow \rightarrow^3$ is always derivable, because a two-armed case is safe for every sum constructor. We include this premise anyway, to highlight the two rules’ similarity.

Several rules are not tied to specific type connectives. An assumption $x : A$ in $\Gamma$ could be read "$x$ synthesizes $A"{}$, so Syn\text{Var} synthesizes its type. Rule Syn\text{Anno} synthesizes the type given in an annotation $[e : A]$, provided $e$ checks against $A$. Following earlier bidirectional systems (Davies and Pfenning 2004, Dunfield and Pfenning 2004), the subsumption rule has a checking conclusion and a synthesizing premise. The checking conclusion ensures that a subsumption, which loses information, is applied only with the programmer’s consent: the type being checked against is derived from a type annotation. The synthesizing premise ensures that we “make progress” as we move from the goal $e \equiv A$ to the subgoal $e \Rightarrow A'$; we cannot use Chk\text{Sub} as the concluding rule of its own premise. In addition to subtyping and change of precision, Chk\text{Sub} with $A = A'$ (using reflexivity) allows us to use a derivation of $\Gamma \vdash e \Rightarrow A$ where we need a derivation of $\Gamma \vdash e \equiv A$. For example, applying a function to a variable requires this rule: Syn\text{Var} synthesizes, but Syn\text{Anno} has a checking premise.

\textbf{Complexity.} Typing in the bidirectional system takes polynomial time. With one exception, the bidirectional rules are in one-to-one correspondence with syntactic forms. The exception is Chk\text{Sub} which can be used to check any synthesizing form. So bidirectional typing is syntax-directed in a slightly looser sense than the usual one: For each pair of a syntactic form and a direction (checking or synthesis), exactly one rule applies; if that rule is Chk\text{Sub} then exactly one rule applies to derive its synthesizing premise. Thus, the size of a derivation (if one exists) is, at most, twice the size of the expression.

\textbf{Variations on a theme.} Several checking rules could be supplemented with a synthesizing rule, or (in the case of Chk\text{Anno}) replaced. A synthesizing version of Chk\text{SumIntro} would be problematic: while we might synthesize the sum constructor $\rightarrow^1$, synthesizing $e$ for $A_1$ tells us only one component of the sum. Our system enjoys uniqueness of synthesis: given $\Gamma$ and $e$, $e$ synthesizes (at most) one type. Synthesizing the other component of the sum would synthesize an infinite number of types. Moreover, a direct implementation would need to guess the other component.

A synthesizing version of Chk\text{SumElim2} would be straightforward; for example, we could synthesize $e_1 \Rightarrow B_1$ and $e_2 \Rightarrow B_2$ and synthesize their join $B_1 \lor B_2$ in the conclusion. Except for Chk\text{UnitIntro} all of these variations—while perhaps convenient in practice—would make the system larger and more complicated. This paper presents a core calculus; we leave exploration of such variations to future work.

\textbf{4.1 Static System} Two restricted versions of the bidirectional system are of interest. The first is a static system: a simply typed $\lambda$-calculus with sums and refinements over sums, without any dynamic sums. The syntax (Figure 8) is the same as the source language, except for $\delta^*$ which
can only be $+$, $+$+, or $+$2. We follow the bidirectional system in deriving rules for sub-sum, subtyping, and typing; the judgments are decorated with $\subseteq$ for “static”. The interesting difference is in the typing rules for sums: the introduction rule checks that the sum is above $+$, (instead of $+$1), and the one-arm elimination (instead of $+$2) checks that the sum is below $+$, (instead of $+$1), that is, the sum is exactly $+$.

4.2 Dynamic System

The static system omits dynamic systems; the dynamic system’s only sum is the dynamic sum $+$?. Since one-armed cases are allowed on type $+$?, this corresponds to datatypes in Standard ML. The metavariables and judgments are decorated with $D$ for “dynamic”. For space reasons, the definition of this system is in the supplementary material (Appendix A).

4.3 Metatheory

The bidirectional system is decidable. The $\delta' \leq \delta$ judgment is immediately decidable (taking the transitive closure of the rules), and the $\Lambda' \leq \Lambda$ judgment is decidable because each rule moves from larger type expressions to smaller ones. The same holds for $\subseteq$, so directed consistency is decidable. The argument for the typing rules is slightly more interesting, as [CheckSub] is a stationary rule (the premise and conclusion type the same expression). However, since this rule moves from checking to synthesis, and no stationary rule moves from synthesis to checking (in [SynAnno] the expression becomes smaller), decidability holds.

**Theorem 1** (Decidability of bidirectional typing).

1. Given $\Gamma$, $e$ and $A$, the judgment $\Gamma \vdash e : A$ is decidable.

2. Given $\Gamma$ and $e$, the judgment $\Gamma \vdash e : A$ is decidable.

The bidirectional system is sound with respect to the type assignment system: if $e$ is well-typed in the bidirectional system, it is well-typed in the type assignment system. (Proofs can be found in the supplementary material.)

**Theorem 2** (Bidirectional soundness).

If $\Gamma \vdash e : A$ or $\Gamma \vdash e : A$ then $\Gamma \vdash e : A$.

The bidirectional system is also complete: given $e : A$ in the type assignment system, it is always possible to add annotations that make $e$ well-typed in the bidirectional system. We write $e \equiv e'$ when $e'$ is the same as $e$ except that $e'$ may have extra annotations.

**Theorem 3** (Annotatability).

If $\Gamma \vdash e : A$ then there exist $e'$ and $e''$ such that (1) $\Gamma \vdash e' : A$ where $e \equiv e'$, and (2) $\Gamma \vdash e'' : A$ where $e \equiv e''$.

We also show that bidirectional typing derivations are robust under imprecision: if $e' \equiv A'$, replacing annotations in $e'$ with more imprecise types preserves typing. This corresponds to part 1 of the gradual guarantee of [Siek et al. 2015, Theorem 5 on p. 11]. An example illustrating this theorem’s significance appears below in Section 4.2.

First, $\Gamma' \subseteq \Gamma$ is defined pointwise. Second, let $e' \subseteq e$ if, for each annotation $(e_0 : A)$ in $e'$, there is a corresponding annotation $(e : A)$ in $e$ where $e' \subseteq e$. (For full inductive definitions, see Figures 15 and 16 in the supplementary material.)

**Theorem 4** (Varying precision of bidirectional typing).

1. If $\Gamma' \subseteq \Gamma$ and $e' \subseteq e$ then $\Gamma' \vdash e : A$.

2. If $\Gamma' \vdash e' : A'$ and $e' \subseteq e$ then $\Gamma' \vdash e : A'$.

The nonempty context is needed for the proof cases for rules whose premises add to $\Gamma'$, such as [CheckSumElim1].
An earlier version of the system, which did not allow gain of precision, has a weaker property: in that system, the given expression \( e \) is not necessarily typable, but there exists some “even more imprecise” expression \( e' \) that is typable. See Theorem 13 in Appendix C.

**Static system.** As the static system is essentially a restriction of the bidirectional system, it is easy to turn a derivation in the static system into a derivation in the bidirectional system; this is the first part of the following theorem.

Completeness is more interesting: Given a bidirectional derivation whose conclusion is static—that is, the context \( \Gamma \), expression \( e \), and type \( \Delta \) are within the restricted static grammar—we can build a derivation in the static system. This holds because of a subformula property: if there are no dynamic sums in \( \Gamma \), \( e \), and \( \Delta \), then dynamic sums cannot appear anywhere in the bidirectional derivation.

**Theorem 5** (Static soundness and completeness).

1. **Soundness:**
   - (a) If \( \Gamma \vdash e^S \leftarrow A^S \) then \( \Gamma \vdash e^S \leftarrow A^S \)
   - (b) If \( \Gamma \vdash e^S \leftarrow A^S \) then \( \Gamma \vdash e^S \leftarrow A^S \).

2. **Completeness:**
   - (a) If \( \Gamma \vdash e^S \leftarrow A^S \) then \( \Gamma \vdash e^S \leftarrow A^S \).
   - (b) If \( \Gamma \vdash e^S \leftarrow A^S \) then \( \Gamma \vdash e^S \leftarrow A^S \).

This theorem directly corresponds to part 1 of Theorem 1 of Siek et al. (2015, p. 9) for “fully annotated” expressions. In that work, an expression is fully annotated if it has no gradual type annotations. In our system, expressions without annotations are static.

A corresponding theorem holds for the dynamic system and, in turn, corresponds to part 1 of Theorem 2 of Siek et al. (2015, p. 9). This is a rough correspondence: in our bidirectional system, dynamism is restricted to sum types and arises only through annotations. See Theorem 15 in the appendix.

### 4.4 Example

To see why Theorem 4 matters, consider the following example. Suppose we want to transform a program that uses dynamic sums into one that uses static sums. The program has a function \( f \) of type \((\text{Unit} + \text{Int}) \rightarrow \text{Int}\), which is called with an argument \( x \) of type \( \text{Unit} + \text{Int}\).

\[
\begin{align*}
\text{let } f &= (\lambda y. \cdots) : (\text{Unit} + \text{Int}) \rightarrow \text{Int} \quad \cdots \\
\text{let } x &= e_x : (\text{Unit} + \text{Int}) \quad \text{in} \\
\text{f x}
\end{align*}
\]

(We assume that \( e_x \) is a checking form that needs an annotation; if \( e_x \) synthesizes \((\text{Unit} + \text{Int})\), the annotation could be removed.) The programmer realizes that if \( f \) and \( x \) both work with a right injection (perhaps its body is a one-armed case on \( \text{inj}_2 \)), and that \( x \) should always be a right injection.

\[
\begin{align*}
\text{let } f &= (\lambda y. \cdots) : (\text{Unit} + \text{Int}) \rightarrow \text{Int} \quad \cdots \\
\text{let } x &= e_x : (\text{Unit} + \text{Int}) \quad \text{in} \\
\text{f x}
\end{align*}
\]

If this program type-checks and contains no remaining dynamic sum annotations, we know that \( f \) and \( x \) actually satisfy their annotations, and that the application \( f \ x \) will not cause any match or cast failures. Theorem 4 says that the annotations can be changed one at a time: the program with \( + \) in the type of \( f \) but \( \ast \) in the type of \( x \) is well-typed, as is the program with \( \ast \) in the type of \( f \) but \( + \) in the type of \( x \):

\[
\begin{align*}
\text{let } f &= (\lambda y. \cdots) : (\text{Unit} + \text{Int}) \rightarrow \text{Int} \quad \cdots \\
\text{let } x &= e_x : (\text{Unit} + \text{Int}) \quad \text{in} \\
\text{f x}
\end{align*}
\]

When synthesizing the type of \( f \), we use \( \text{ChkCSub} \) to gain precision in \( x \):

\[
\begin{align*}
\Gamma \vdash f &\Rightarrow (\text{Unit} + \text{Int}) \quad \text{ChkCSub} \\
(\text{Unit} + \text{Int}) &\to (\text{Unit} + \text{Int}) \quad \text{Syn} \to \text{Elim} \\
\Gamma \vdash x &\Rightarrow (\text{Unit} + \text{Int}) \\
\end{align*}
\]

A precise annotation that differs from the correct one, such as \( \text{Unit} + 1 \) Int on \( x \), may cause an error—either at type-checking time, or at run time. But a precise annotation that is correct will not cause an error, and constitutes a step towards a completely static program.

### 5. Target Language and Translation

#### 5.1 Target Syntax and Semantics

\[
\begin{align*}
i &\ ::= \ 1 \ | \ 2 \\
\text{Target sums} &\ ::= \ f \ | \ 0 \\
\text{Target terms} &\ ::= \ \text{O} \ | \ \lambda x. M \ | \ M_1 \ M_2 \ | \ \text{inj}_1 M \\
&\quad | \ \text{case}(M, \text{inj}_1 x_1, M_1, \text{inj}_2 x_2, M_2) \\
&\quad | \ \text{case}(M, \text{inj}_1 x, M_1) \\
&\quad | \ (\text{inj}_1 \ M_1 \ | \ M \ | \ \text{matchfail}) \ |
\end{align*}
\]

\[
\begin{align*}
\text{Values} &\ ::= \ \text{O} \ | \ \lambda x. M \ | \ \text{inj}_1 W \\
\text{Target types} &\ ::= \ \text{Unit} \ | \ T_1 T_2 \ | \ T_1 \to T_2 \\
\text{Target typing contexts} &\ ::= \ \emptyset \ | \ \Theta, x : T
\end{align*}
\]

**Figure 9.** Target syntax

Our target language is a statically typed \( \lambda \)-calculus with static sum types and a cast construct. The syntax is shown in Figure 9. We write \( M \) for target terms (expressions), \( W \) for values, and \( T \) for target types. The target sum constructors are all the static sum types from the source language: \( +, \ast, 1 \), and \( + \). In addition, we have a cast construct \( \phi_2 \subseteq \phi_1 \ M \), which casts from sum \( \phi_1 \) to \( \phi_2 \). A failing cast, such as \( \phi_2 \subseteq \phi_1 \ [\text{inj}_1, \ O] \), steps to the error term \( \text{matchfail} \).

Much of the target type system (Figure 10) follows the source type assignment system, if that system were restricted to static sum types. Since the target lacks any dynamic sum constructors (like \( + \)), target subtyping says only that \( + \) and \( \ast \) are subtypes of \( + \); this corresponds to datasset refinement systems, where every datasset is a subsort of a “top” datasset for the type being refined. Our type-directed translation (Section 5.2) transforms the gradual property of types into dynamic checks at the term level; rule \( \text{TCast} \) casts between sum constructors, and rule \( \text{TMatchfail} \) gives any type to matchfail, which represents the failure of a cast.

Our target language (Figure 11) has a standard call-by-value small-step semantics, extended with casts. Evaluation contexts \( E \) are terms with a hole \( [] \), where the hole represents a term in an evaluation position: if target term \( M = E[M_0] \), and \( M_0 \) reduces—written \( M_0 \to R M_0' \)—then the larger term \( M \) steps to \( E[M_0'] \).

The cast reduction rules represent the three relevant situations: (1) an \( \text{upcast} \) to a supertype succeeds (\( \text{ReduceUpcast} \)), (2) a downcast from \( + \) to \( \ast \), succeeds if \( i \) matches the injection \( \text{ReduceCastSuccess} \), (3) a downcast from \( + \) to \( + \), fails, reducing to \( \text{matchfail} \), if \( i \) doesn’t match the injection \( \text{ReduceCastFailure} \).

#### 5.2 Type-Directed Translation \( \rightsquigarrow \)

To translate source programs into target programs with explicit casts between sum types, we use a judgment \( \Gamma \vdash e : A \rightsquigarrow M \). Most of the rules (in Figure 12) follow the type assignment rules, with the addition of \( \rightsquigarrow M \). Given \( e \) of type \( A \), the rules produce a target term \( M \) of type \( T \) where \( T \) is the translation of \( A \), written \( \lbrack A \rbrack \).
| \( \phi' \leq \phi \) | Sum \( \phi' \) is a subsum of \( \phi \) |
| \( \phi \leq \phi \) | \( +_1 \leq + \) |
| \( T' \leq T \) | Target type \( T' \) is a subtype of \( T \) |

\[ \Theta \vdash M : T \] Under context \( \Theta \), target term \( M \) has target type \( T \)

| \( (T_1 \phi' T_2) \leq (T_1 \phi T_2) \) | (Unit \leq \Unit) |
| \( T_1 \leq T_1 \) \( T_2 \leq T_2 \) \( \phi' \leq \phi \) | \( (T_1 \phi' T_2) \leq (T_1 \phi T_2) \) |

\[ \Theta(x) = T \] \( \Theta \vdash x : T \) \( \text{TVar} \)

\[ \Theta \vdash M : T' \] \( T' \leq T \) \( \Theta \vdash M : T \) \( \text{TSub} \)

\[ \Theta \vdash \text{matchfail} : T \] \( \Theta \vdash () : \text{Unit} \) \( \text{TMatchfail} \)

\[ \Theta \vdash M_0 : T_1 + T_2 \] \( \Theta, x : T_1 \vdash M : T \) \( \Theta \vdash \text{case}(M_0, \text{inj}_1, x, M) : T \) \( \text{T+Intro} \)

\[ \Theta, x : T_1 \vdash M : T_2 \] \( \Theta \vdash \lambda x, M : (T_1 \rightarrow T_2) \rightarrow T \rightarrow \text{Intro} \)

\[ \Theta \vdash M_0 : T_1 + T_2 \] \( \Theta, x_1 : T_1 \vdash M_1 : T \) \( \Theta, x_2 : T_2 \vdash M_2 : T \) \( \Theta \vdash \text{case}(M_0, \text{inj}_1, x_1, M_1, \text{inj}_2, x_2, M_2) : T \) \( \text{T+Elim} \)

\[ \Theta \vdash M_1 : T' \rightarrow T \] \( \Theta \vdash M_2 : T' \rightarrow T \rightarrow \text{Elim} \)

\[ \Theta, x : T_1 \vdash M : T_2 \] \( \Theta \vdash \lambda x, M : (T_1 \rightarrow T_2) \rightarrow T \rightarrow \text{Intro} \)

\[ \text{Evaluation contexts} \]

\[ \mathcal{E} ::= [] \]

\[ \text{| } \ | \inj_1 \mathcal{E} \]

\[ | \text{case}(\mathcal{E}, \inj_1, x, M) \]

\[ | \text{case}(\mathcal{E}, \inj_1, x_1, M_1, \inj_2, x_2, M_2) \]

\[ | (\phi \Leftarrow \phi') \mathcal{E} \]

\[ | \mathcal{E} \mathcal{M}_1 \mathcal{W}_1 \mathcal{E} \]

\[ \text{Target term } M \text{ reduces to } M' \]

\[ \mathcal{M} \mapsto_R \mathcal{M}' \]

\[ \langle \phi \Leftarrow \phi' \rangle \mathcal{W} \mapsto_R \mathcal{W} \text{ where } \phi' \leq \phi \]

ReduceUpcast

ReduceCastSuccess

ReduceCastFailure

ReduceCase1

ReduceCase2

Reduceβ

\[ \mathcal{M} \mapsto_R \mathcal{M}' \]

\[ \mathcal{E}[\mathcal{M}] \mapsto \mathcal{E}[\mathcal{M}'] \]

StepContext

\[ \mathcal{E} \neq [] \]

\[ \mathcal{E}[\text{matchfail}] \mapsto \text{matchfail} \]

StepMatchfail

This translation (Figure 12) top) maps the source sums + and + to the target sum +, and maps the other source sums to +.

We extend type assignment, rather than the bidirectional system, because translation should be independent of bidirectionality: Type assignment is stable under variations in the bidirectional “recipe”, so if we decided to synthesize a type for (), we could leave the translation untouched. That said, an implementation would be based on a bidirectional version of the translation—replacing “\( \Leftarrow \)” with “\( \Leftarrow \)” or “\( \Rightarrow \)”, following Figure 7.

The interesting translation rule is \text{STCSub}, which inserts a coercion context \( \mathcal{C} \). This context coerces between two direction-consistent types, so it comes up to three coercions (cf. Figure 5): from a more imprecise type to a less imprecise type, from that type to a supertype, and from the supertype to a more imprecise type.

Our coercion judgment \( \mathcal{A} \Rightarrow A \Leftarrow C \) produces a context \( \mathcal{C} \), a target type containing a hole such that, if \( M \) has type \( T' = [A'] \), then \( C[M] \) has type \( T = [A] \). Rule \text{CoeUnit} produces a hole, which behaves as the identity function. Rule \text{Coe} produces a function: given a hole \( [] \) filled by a function of type \( T_1 \rightarrow T_2 \), it constructs \( \lambda x, C_2 \{ [] \ C_1[x] \} \). This function has type \( T_1 \rightarrow T_2 \): it applies cast \( C_1 \) to \( x \), yielding a value of type \( T'_1 \). Applying the original function yields an \( T'_2 \), which cast \( C_2 \) transforms into an \( T_2 \).

Three rules generate coercions between sum types: \text{CoeCase1L}, \text{CoeCase1R} and \text{CoeCase2}. The first two rules handle sums that are definitely a left injection, or definitely a right injection: we apply \text{CoeCase1L} whenever we are coercing from \( A'_1 \delta A'_2 \) where \( \delta' \) is \( +_1 \) or \( +_2 \), and \text{CoeCase1R} when \( \delta' \) is \( +_1 \) or \( +_2 \).

In \text{CoeCase1L} we recursively generate a coercion \( C_1 \) from \( A'_1 \), and a cast \( C_3 \) from \( \delta' \). The conclusion generates a coercion by matching the given value (replacing \( [] \) against \( \inj_1 \) \( x_1 \), constructing \( \inj_1 \{ C_1[x_1] \} \)), to which we apply \( C_3 \). \text{CoeCase1} is symmetric.

\text{CoeCase2} handles the cases not covered by the previous two rules. In addition to doing the work of the previous two rules, it generates casts \( C'_1 \) and \( C'_2 \), applying them in each arm. According to \text{STSumIntro}, an injection \( \inj_1 \) has a type whose sum constructor is \( +_1 \), so \text{CoeCase2} applies \( C'_1 \) which takes \( +_1 \) to \( \delta' \). Similarly, the rule applies \( C'_2 \), which takes \( +_2 \) to \( \delta' \). Since \text{CoeCase2} applies \( C_3 \) (from \( \delta' \) to \( \delta \)) to the entire case, the result will be \( \delta \).

5.3 Target Precision

We will prove that more precise source typings—differently annotated versions of the same source expression—produce more precise target terms. We will also prove that precision of the target terms is preserved by stepping, and that if a more precise target term converges (steps to a value), so does a less precise target term.
Coercion $C$ coerces sum $\delta'$ to sum $\delta$:

$$\delta' \Rightarrow \delta \leftarrow C$$

CoeCast

$$\delta' \Rightarrow \delta \leftarrow C \leq \delta'$$

CoeSub

Given two terms with casts $M' = \langle \phi_2 \leftarrow \phi_1 \rangle$ and $M = \langle \phi_2 \leftarrow \phi_1 \rangle$, we will consider $M'$ more precise than $M$ if the cast in $M'$ is more precise: $\langle \phi_2 \leftarrow \phi_1 \rangle \Rightarrow \langle \phi_2 \leftarrow \phi_1 \rangle$. Let $ac$ be a cast; it must be either a safe cast $ac$ like $(+ \leftarrow +)$ or $\langle + \leftarrow + \rangle$, a backward cast $bc$ of the form $\langle + \leftarrow + \rangle$, or a (doomed) match-failure cast $mc\langle + \leftarrow + \rangle$ or $\langle + \leftarrow + \rangle$. These are classified by the grammar in Figure 13.

Equal casts should be equally precise, so rule $\text{Cast=Ref}$ makes the relation $ac \Rightarrow ac$ reflexive. Following the idea that the more precisely typed term should fail more, a safer cast should be less precise; this leads to $\text{CastM} \leq \text{CastM'}$ and $\text{CastM} \leq \text{S}$.

The other rules are subtle. They compare particular safe casts and/or backward casts, relying implicitly on typing. For example, the last rule says (with $i = 1$) that $\langle + \leftarrow + \rangle \Rightarrow \langle + \leftarrow + \rangle$. We will ultimately need to show that if the cast on the left succeeds, so does the cast on the right. The left-hand cast is $\langle + \leftarrow + \rangle$, which always succeeds. The right-hand cast succeeds if it is given $\text{inj}_1$. If the value being cast is well-typed, then (by $\text{FTCast}$) it will indeed have type $\langle + \leftarrow + \rangle$.

Finally, note that a more precise source type may result in a one-armed case in a coercion, while the less precise typing results in a two-armed case. For example, $\langle + \rangle \leftarrow \langle + \rangle$ is less precise than $\langle + \rangle$.
5.4 Metatheory

The target system satisfies preservation and progress:

**Theorem 6** (Type preservation).

\( \vdash M : T \) and \( M \rightarrow M' \) then \( \vdash M' : T \).

**Theorem 7** (Progress).

\( \vdash M : T \) then either (a) \( M \) is a value, or (b) there exists \( M' \) such that \( M \rightarrow M' \), or (c) \( M \) is matchfail.

By itself, the above progress statement leaves open the possibility that a well-typed target term \( M' \) will step to matchfail. However, if \( M \) has no casts, it will not step to matchfail.

**Theorem 8** (matchfail-freeness).

If \( M \) is cast-free and matchfail-free and \( M \rightarrow M' \) then \( M' \) is cast-free and matchfail-free.

For cast-free terms, combining Theorems 7 and 8 gives a version of progress without the possibility of match failure.

**Corollary.** If \( M \) is cast-free and matchfail-free and \( \vdash M : T \) then either (a) \( M \) is a value, or (b) there exists \( M' \) such that \( M \rightarrow M' \).

We also prove that the translation takes well-typed source programs to well-typed target programs. The theorem takes a type assignment derivation, but Theorem 8 can produce such a derivation from a bidirectional typing derivation.

**Theorem 9** (Translation soundness).

If \( \Gamma \vdash e : A \) then there exists \( M \) such that \( \Gamma \vdash e : A \leftarrow M \) and \( \Gamma \vdash M : A \).

The proof relies on several lemmas, e.g. that the generated coercions \( C \) are well-typed; see the supplementary material.

A great advantage of static typing is that, for a suitable definition of "wrong", static programs don’t go wrong. The theorem below proves that translating a static program yields a target term \( M \) that has no casts; by Theorem 8, \( M \) will never step to matchfail.

**Theorem 10** (Static derivations don’t have match failures).

If \( \Gamma \vdash e : A \) then there exists \( M \) such that \( \Gamma \vdash e : A \leftarrow M \) and \( M \) is free of casts and matchfail.

Together, preservation and progress correspond to Theorem 3 (type safety) of Siek et al. [2015, p. 9]. Their blame-subtyping Theorem 4 says that safe casts (casts from a subtype to a supertype) cannot be blamed (cannot fail); our translation does not insert safe casts at all, and our Theorem 10 shows that expressions without dynamic sums produce target terms without casts.

The remaining results concern precision. We show that more precise annotations translate to more precise terms, that target precision is preserved by stepping, and that if a target term converges, then a less precise version also converges.

We must note that the first of these results, Theorem 11, uses a modified version of the translation: one that always inserts casts, even safe ones; this simplifies part of the proof. In effect, the modified translation (Figure 21 in the appendix) does not have rule CoercCast and always uses rule CoercRef. Similarly, we modify CoercCase 1 and CoercCase 2 to always insert casts within each arm, like \( C_1' \) and \( C_2' \) in CoercCase 2. Since the only difference is the presence of casts that cannot fail, the terms generated by either translation must both step to the same value, or both generate matchfail.

**Theorem 11** (Translation preserves precision).

Suppose \( \Gamma' \subseteq \Gamma \) and \( e' \subseteq e \).

1. If \( \Gamma' \vdash e' : A' \leftarrow A \) and \( \Gamma \vdash e : A \) then \( \Gamma' \vdash e' : A' \leftarrow M' \) and \( \Gamma \vdash e : A \leftarrow M \) where \( M' \subseteq M \).

2. If \( \Gamma' \vdash e' : A' \leftarrow A \) then \( \Gamma' \vdash e' : A' \leftarrow M' \) and \( \Gamma \vdash e : A \leftarrow M \) where \( A' \subseteq A \) and \( M' \subseteq M \).

**Theorem 12** (Stepping preserves precision).

If \( \vdash M_1 : T_1 \) and \( \vdash M'_1 : T_1 \) then either (a) \( M_1 \) is a value and \( M'_1 \subseteq M_1 \), or (b) there exists \( M_2 \) such that \( M_1 \rightarrow M_2 \) and \( M'_1 \subseteq M_2 \), or (c) \( M_1 \) is matchfail and \( M'_1 \subseteq M_1 \).
Definition 1. A closed term $M$ converges if $M \rightarrow^* W$ for some value $W$, and diverges if the stepping sequence never terminates.

Note that matchfail neither converges nor diverges, and that divergence is not possible in our language.

Theorem 13 ($\preceq$ respects convergence). If $M' \preceq M$ where $\cdot \vdash M' : T'$ and $\cdot \vdash M : T$ and $M'$ converges then $M$ also converges.

If $M' \preceq M$ and they converge to injections $inj_j W'$ and $inj_k W$ then Theorem [13] gives $inj_j W' \preceq inj_k W$. By inversion on the definition of $\preceq$, we have $i = k$. Similar results would hold if $\preceq$ were extended for base types.

Together with Theorem [11] this means that if we translate two source expressions $e' \preceq e$ to $M'$ and $M$, and $M'$ converges to a value of base type, $M$ will converge to the same value. This corresponds to Theorem 5 (gradual guarantee), part 2, of [Siek et al. 2015].

6. Related Work

Sums and refinements. Sum types are well-established in a variety of programming languages, though practical languages tend to embed them within larger mechanisms: ML datatypes can encode sums, but also recursion. Refinement type systems, such as data-sort refinements [Freeman and Pfenning 1991; Davies 2005] and indexed types [Xi and Pfenning 1999], have been built on these larger mechanisms. This gives a close connection to practice, but needs additional machinery such as constructor types and signatures. Such machinery is not central to our investigation; in contrast, we distill data-sort refinements to one essential feature: distinguishing whether we have a left or right injection.

These systems often have a refinement relation $\sqsubseteq$: if $A$ is a sort (refined type) and $\tau$ is an unrefined type, $A \sqsubseteq \tau$ says that $A$ refines $\tau$. Both the symbol and the high-level concept resemble our relation $A' \sqsubseteq A$, but the refinement relation is more rigid: it cannot compare two sorts, or two unrefined types, and it certainly cannot derive $(A_1 \rightarrow A) \sqsubseteq (A_1 \rightarrow \tau)$, where $(A_1 \rightarrow \tau)$ mixes a refined type $A_1$ with an unrefined type $\tau$. Nonetheless, the covariance of this relation on function types—in contrast to subtyping, which must be contravariant—made us more confident that our precision relation should be covariant.

Koot and Hage [2013] formulate a constraint-based type system that analyzes pattern matches, using a characterization of data somewhat reminiscent of data-sort refinements. Their system needs no type annotations, but is (necessarily) incomplete.

Gradual typing. Our approach to expressing uncertainty in a type system was inspired by gradual typing, introduced by Siek and Taha [2009], in which $\false$ (often written $\star$) is an uncertain type (it could be Int, a function type, or anything else). We confine uncertainty to refinement properties of sum types, making the effect on the overall type system less dramatic; still, several mechanisms of gradual typing appear in our work. For example, we also have precision relations on types and (through annotations) expressions.

Our directed consistency is somewhat similar to consistent subtyping for gradual object-based languages [Siek and Taha 2007]. Consistent subtyping augments subsumption with consistent equality (roughly, gain and loss of precision) on either the subtype or supertype, but not both. Drawing on abstract interpretation, Garcia et al. [2016] give a different but equivalent formulation of consistent subtyping. In these systems, the underlying subtyping relation is defined over static types only. Allende et al. [2014] also have a notion of directed consistency, but the connection to our relation is less clear.

Siek et al. [2015] propose several criteria as desirable for gradual type systems. We prove properties that correspond to some of their criteria: Theorems [5] and [13] correspond to the first parts of Theorems 1 and 2 of [Siek et al. 2015], our Theorem [10] corresponds to their Theorem 4, our Theorem [5] corresponds to part 1 of their Theorem 5 (gradual guarantee), and our Theorems [11] and [13] corresponds to part 2 of their Theorem 5.

Some systems of gradual typing include a notion of blame [Wadler and Findler 2009], associating program labels to casts so that a failing cast “blames” some program location. It may be possible to incorporate blame into our approach; we omit it to focus on other issues.

We are not the first to apply ideas from gradual typing to less traditional areas: for example, Bañados Schweter et al. [2014] develop a gradual effect system, and McDonell et al. [2016] develop a tool for moving between ADTs and more precise GADTs.

Bidirectional typing. Originating as folklore and first discussed explicitly by Pierce and Turner [1998], bidirectional typing has been used extensively in type systems for which full inference is undecidable or otherwise problematic [Freeman and Pfenning 1991; Coquand 1996; Xi and Pfenning 1999; Davies and Pfenning 2003; Pientka 2008]. A strength of many bidirectional type systems, sometimes overlooked, is that they have some variety of sub-formula property. In some systems, this property serves to make type checking more feasible—for example, for Davies [2003] and Dunfield [2007], it controls the spread of intersection types. For Dunfield [2013], where evaluation order is implicit in terms and explicit in types, it prevents the spontaneous generation of by-name types; in our system, it prevents the spontaneous generation of gradual sum types.

The gradual type system of Garcia and Cinina [2015, p. 306] is not bidirectional, but enjoys a similar property: “dynamically the uncertain type $\star$ is introduced only via program annotations”. However, their rules can be viewed as a bidirectional system that always synthesizes, except at annotations.

7. Future Work

We plan to implement the bidirectional type system, which will allow us to test whether our approach is practical. We are particularly interested in whether our formulation of precision, combined with the annotation discipline of bidirectional typing, strikes a good balance: the annotation burden should be reasonable, but imprecision should not appear out of nowhere. Also, it is unclear whether programmers would have any use for the sum types $\star$ and $\ast$; if not, error messages should read “expected $+$ or $\ast$” rather than “expected $\star$”, for example.

We would also like to enrich the language with intersection types, recursive types, and polymorphism. Intersection types are important for data-sort refinements: for example, if we encode booleans as Unit + Unit, the data sorts True and False are Unit + Unit and Unit + Unit + Unit. Then negation should have type (True $\rightarrow$ False) $\cap$ (False $\rightarrow$ True). We also want to evaluate the run-time efficiency of coercions—a common concern in gradual type systems.

Acknowledgments

We would like to thank Ronald Garcia, Felipe Bañados Schweter, Joey Eremondi, Rui Ge, Jodi Spacek, Alec Thériault, and the anonymous reviewers for their feedback on several versions of this work.

References


Appendix to “Sums of Uncertainty: Refinements go gradual” (POPL 2017)

A. Dynamic System

Dynamic expressions
\[ e^D := () \mid x \mid \lambda x. e^D \mid e^D \oplus e^D \mid \text{inj}_1 e^D \mid (e^D : A^D) \]
\[ \mid \text{case}(e^D, \text{inj}_1 x_1. e^D, \text{inj}_2 x_2. e^D) \mid \text{case}(e^D, \text{inj}_1 x. e^D) \]

Dynamic types
\[ A^D := \text{Unit} \mid A^D + A^D \mid A^D \rightarrow A^D \]

Dynamic typing contexts
\[ \Gamma^D := \cdot \mid \Gamma, x : A^D \]

\[ \Gamma^D \vdash_D e^D \iff A^D \]

\[ \Gamma^D \vdash_D e^D \Rightarrow A^D \]

Under typing context \( \Gamma^D \), expression \( e^D \) checks against type \( A^D \)

Under typing context \( \Gamma^D \), expression \( e^D \) synthesizes type \( A^D \)

\[ \begin{align*}
\Gamma^D(x) &= A^D \\
\Gamma^D \vdash_D x \Rightarrow A^D &\text{ DVar} \\
\Gamma^D \vdash_D e^D \Rightarrow A^D &\text{ DSub} \\
\Gamma^D \vdash_D (e^D : A^D) \Rightarrow A^D &\text{ DAnno} \\
\Gamma^D \vdash_D () \Rightarrow \text{Unit} &\text{ DUnitIntro} \\
\Gamma^D, x : A^D \vdash_D e^D \Rightarrow A^D &\text{ DIntro} \\
\Gamma^D \vdash_D e^D \Rightarrow A^D &\text{ D→Intro} \\
\Gamma^D \vdash_D e_1 \oplus e_2 \Rightarrow A^D &\text{ D+Intro} \\
\Gamma^D \vdash_D \text{case}(e_0^D, \text{inj}_1 x. e^D) \Rightarrow A^D &\text{ D+Elim1} \\
\Gamma^D \vdash_D \text{case}(e_0^D, \text{inj}_1 x_1. e_1^D, \text{inj}_2 x_2. e_2^D) \Rightarrow A^D &\text{ D+Elim2}
\end{align*} \]

Figure 14. The dynamic system: the bidirectional system restricted to \( +^2 \).

B. Omitted Definitions

\[ e' \sqsubseteq e \]
Expression \( e' \) is more precise than \( e \)

\[ \begin{array}{c}
\emptyset \sqsubseteq \emptyset \\
x \sqsubseteq x \\
\lambda x. e' \sqsubseteq \lambda x. e \\
e_1' \sqsubseteq e_1 \\
e_2' \sqsubseteq e_2 \\
ed_1' \sqsubseteq e_1 \\
ed_2' \sqsubseteq e_2 \\
\text{case}(e_0', \text{inj}_1 x_1. e_1, \text{inj}_2 x_2. e_2') \sqsubseteq \text{case}(e, \text{inj}_1 x_1. e_1, \text{inj}_2 x_2. e_2) \\
e_1' \sqsubseteq e_1 \\
e_2' \sqsubseteq e_2 \\
\text{case}(e_0', \text{inj}_1 x. e') \sqsubseteq \text{case}(e, \text{inj}_1 x. e)
\end{array} \]

\[ \Gamma' \sqsubseteq \Gamma \]
Typing context \( \Gamma' \) is more precise than \( \Gamma \)

\[ \begin{array}{c}
\vdash_e \\
\Gamma' \sqsubseteq \Gamma \\
\Gamma', x : A' \sqsubseteq \{ \Gamma, x : A \}
\end{array} \]

Figure 15. Precision on expressions and contexts

Several results involve precision of expressions and typing contexts, shown in Figure 15 these are the straightforward lifting of type precision (Figure [4]).

\[ e' ::= e \]
Expression \( e' \) is annotation-ly equivalent to \( e \)

\[ \begin{array}{c}
\emptyset ::= \emptyset \\
x ::= x \\
e' ::= e \\
e' ::= \{e : A\} \\
\lambda x. e' ::= \lambda x. e \\
e_1' ::= e_1 \\
e_2' ::= e_2 \\
ed_1' ::= e_1 \\
ed_2' ::= e_2 \\
\text{case}(e_0', \text{inj}_1 x_1. e_1, \text{inj}_2 x_2. e_2') \ ::= \text{case}(e, \text{inj}_1 x_1. e_1, \text{inj}_2 x_2. e_2) \\
e_1' ::= e_1 \\
\text{case}(e_0', \text{inj}_1 x. e') \ ::= \text{case}(e, \text{inj}_1 x. e)
\end{array} \]

Figure 16. Annotation equivalence
C. Differences from the Original Version

The paper that was submitted to POPL differs in two important ways from the final version.

No directed consistency. In the final version, \text{ChkSub}, \text{SCSub} etc. allow (a) gain of precision, (b) subtyping, and (c) loss of precision, formulated via directed consistency. In contrast, the original system had (in each system) two rules: one rule that allowed subtyping (exactly like a traditional subsumption rule), and one rule that allowed loss of precision. For example, the bidirectional system had

\[
\Gamma \vdash e \Rightarrow A' \quad A' \subseteq A \quad \text{**ChkSub}
\]
\[
\Gamma \vdash e \Rightarrow A' \quad A' \supseteq A \quad \text{**ChkImp}
\]

These rules could not type the same expression without an extra annotation (to transition from the checking conclusion of one rule to the synthesizing conclusion of the other).

Moreover, there was no rule to gain precision. In a traditional gradual type system, this would be completely untenable: the point of the “unknown type” in a gradual system is that it can be downcasted to a static type. In the previous version of our system, programmers could write coercions “by hand”: “unknown type” in a gradual system is that it can be downcasted to a static type. In the previous version of our system, programmers could

In Section 2, we explained why \( +_1 \subseteq +_1 \) doesn’t make sense. We also argued against \( +_1 \subseteq +_1 \), on the basis that in directed consistency \( \text{SCSub} \) one could gain precision from \( +_1 \) to \( +_1 \), then use subtyping from \( +_1 \) to \( +_2 \). In the old system, there was no gain of precision, and even loss of precision could not be combined with subtyping (without extra annotation). Thus, we saw no clear property only provides that some expression \( e_j \), which could be more imprecise than \( e \), is well typed.

Different definition of imprecision. In Section 2 we explained why \( +_1 \subseteq +_1 \) doesn’t make sense. We also argued against \( +_1 \subseteq +_1 \), on the basis that in directed consistency \( \text{SCSub} \) one could gain precision from \( +_1 \) to \( +_1 \), then use subtyping from \( +_1 \) to \( +_2 \). In the old system, there was no gain of precision, and even loss of precision could not be combined with subtyping (without extra annotation). Thus, we saw no clear argument against \( +_1 \subseteq +_1 \), and included it in the relation. However, in the absence of gain of precision, the only way the type system could use this was by moving from \( +_1 \) to \( +_1 \), which was also possible via subtyping.

![Figure 17. Original, obsolete definition of precision](image)

\[+_1 \rightarrow +_1 \rightarrow +_2 \rightarrow +_2\]

C.1 Original, weak version of varying precision

Theorem 14 does not hold for the original system. Instead, the following holds, where \( e \subseteq e_j \) means that \( e_j \) is a version of \( e \) with more-imprecise annotations (like \( e' \subseteq e \)) and extra annotations. For example, \( x : \{ x : \text{A} \} \).

**Theorem 14** (Weak version of varying precision).

1. If \( \Gamma \vdash e' \Leftrightarrow A' \) and \( e' \subseteq e \) and \( \Gamma' \subseteq \Gamma \)
   then there exist \( e_j \) and \( A \) such that \( \Gamma \vdash e_j \Leftrightarrow A \) and \( e \subseteq e_j \) and \( A' \subseteq A \).

2. If \( \Gamma \vdash e' \Rightarrow A' \) and \( e' \subseteq e \) and \( \Gamma' \subseteq \Gamma \)
   then there exist \( e_j \) and \( A \)
   such that \( \Gamma \vdash e_j \Rightarrow A \) and \( e \subseteq e_j \) and \( A' \subseteq A \).

Given \( e' \subseteq e \), this weak version of varying precision yields some \( e_j \) that may be more imprecise, \( e' \subseteq e \). This is needed because—in the absence of \( \text{ChkSub} \) which allows precision to be adjusted whenever subsumption is used—a more imprecise annotation may require changing other annotations to make them more imprecise. For example, suppose we are given

\[
e' = ((\lambda x. (x : \text{B} +_2 \text{B})) : (\text{B} +_2 \text{B}) \rightarrow (\text{B} +_2 \text{B}))
\]
\[
e = ((\lambda x. (x : \text{B} +_2^7 \text{B})) : (\text{B} +_2^7 \text{B}) \rightarrow (\text{B} +_2^7 \text{B}))
\]

We can synthesize \( A' = (\text{B} +_2 \text{B}) \rightarrow (\text{B} +_2 \text{B}) \) for \( e' \), but not for \( e \), because the inner annotation on \( x \) makes the \( \lambda \) fail to check against the outer annotation. But we can produce \( e_j = ((\lambda x. (x : \text{B} +_2^7 \text{B})) : (\text{B} +_2 \text{B}) \rightarrow (\text{B} +_2^7 \text{B})) \). Now the uses of \( +_2^7 \) match, and \( e_j \) synthesizes \( A = (\text{B} +_2 \text{B}) \rightarrow (\text{B} +_2^7 \text{B}) \). The remaining \( +_2^7 \) is okay, because of \( \text{**ChkImp} \) in \( (x : \text{B} +_2^7 \text{B}) \), we have \( \Gamma(x) = \text{B} +_2 \text{B} \), which is less imprecise than \( \text{B} +_2^7 \text{B} \).
D. Proofs

D.1 Source System

D.1.1 Subtyping

Lemma 1 (Subtyping inversion).
1. If Unit ≤ A then A = Unit.
2. If A' ≤ Unit then A' = Unit.
3. If A_1' ≤ A and A_2' ≤ A then A = A_1 δ A_2 where A_1' ≤ A_1 and A_2' ≤ A_2 and δ ≤ δ.
4. If A' ≤ A then A' = A where A_1' ≤ A_1 and A_2' ≤ A_2 and δ ≤ δ.
5. If A_1' → A_2' ≤ A then A = A_1 → A_2 where A_1' ≤ A_1 and A_2' ≤ A_2.
6. If A' ≤ A_1 → A_2 then A' = A_1' → A_2' where A_1 ≤ A_1' and A_2 ≤ A_2.

Proof:
1. By case analysis on Unit ≤ A.
   • Case Unit ≤ Unit: Immediate that A = Unit.
2. Symmetric to the previous statement, hence omitted.
3. By case analysis on A_1' ≤ A_2' ≤ A.
   • Case A_1' ≤ A_2' ≤ A_1 δ A_2: Immediate as A = A_1 δ A_2 and subderivations are A_1' ≤ A_1 and A_2' ≤ A_2 and δ ≤ δ.
4. Symmetric to the previous statement, hence omitted.
5. By case analysis on A_1' → A_2' ≤ A.
   • Case A_1' → A_2' ≤ A_1 → A_2: Immediate as A = A_1 → A_2 and subderivations are A_1' ≤ A_1 and A_2' ≤ A_2.
6. Symmetric to the previous statement, hence omitted.

Lemma 2 (Reflexivity of subtyping).
For all types A, it is the case that A ≤ A.

Proof. By induction on the structure of A.

• Case A = Unit: By the definition of precision, A ≤ A.
• Case A = A_1 δ A_2: By the induction hypothesis, A_1 ≤ A_1 and A_2 ≤ A_2. By the reflexivity of subsum, δ ≤ δ. Thus, by the definition of subtyping, A ≤ A.
• Case A = A_2 → A_2: By the induction hypothesis, A_1 ≤ A_1 and A_2 ≤ A_2. Thus, by the definition of subtyping, A ≤ A.

Lemma 3 (Transitivity of subtyping).
If A_1 ≤ A_2 and A_2 ≤ A_3 then A_1 ≤ A_3


• Case A_2 = Unit:
  A_1 ≤ Unit Given
  Unit ≤ A_3 Given
  A_1 = Unit By Lemma[Subtyping inversion]
  A_3 = Unit By Lemma[Subtyping inversion]
  Unit ≤ Unit By Lemma[Reflexivity of subtyping]
  A_1 ≤ A_3 Equivalent

• Case A_2 = A_12 δ A_22:
  A_1 ≤ A_12 δ A_22 Given
  A_1 = A_11 δ A_21 By Lemma[Subtyping inversion]
  A_11 ≤ A_12 ""
  A_21 ≤ A_22 ""
  δ_1 ≤ δ_2 ""
  A_12 ≤ A_22 ≤ A_3 Given
  A_3 = A_13 δ A_23 By Lemma[Subtyping inversion]
  A_12 ≤ A_13 ""
  A_22 ≤ A_23 ""
  δ_2 ≤ δ_3 ""
A_{11} \leq A_{13} \quad \text{By the induction hypothesis}
A_{21} \leq A_{23} \quad \text{By the induction hypothesis}
\delta_1 \leq \delta_3 \quad \text{By the transitivity of} \leq
A_{11} \delta_1 A_{21} \leq A_{13} \delta_1 A_{23} \quad \text{By the definition of} \leq
A_1 \leq A_3 \quad \text{Equivalent}

• Case \( A_2 = A_{12} \rightarrow A_{22} \):

\begin{align*}
A_1 &\leq A_{12} \rightarrow A_{22} \quad \text{Given} \\
A_1 &= A_{11} \rightarrow A_{21} \quad \text{By Lemma 1 (Subtyping inversion)} \\
A_{12} &\leq A_{11} \quad \text{"} \\
A_{21} &\leq A_{22} \quad \text{"}
\end{align*}

A_{12} \rightarrow A_{22} \leq A_3 \quad \text{Given}
A_3 = A_{13} \rightarrow A_{23} \quad \text{By Lemma 1 (Subtyping inversion)}
A_{13} \leq A_{12} \quad \text{"}
A_{22} \leq A_{23} \quad \text{"}
A_{13} \leq A_{11} \quad \text{By the induction hypothesis}
A_{21} \leq A_{23} \quad \text{By the induction hypothesis}
A_{11} \rightarrow A_{21} \leq A_{13} \rightarrow A_{23} \quad \text{By the definition of} \leq
A_1 \leq A_3 \quad \text{Equivalent}

\square

D.1.2 Precision

Lemma 4 (Precision inversion).
1. If Unit \( \subseteq A \) then A = Unit.
2. If A' \( \subseteq \text{Unit} \) then A' = Unit.
3. If A', \delta' \subseteq A then A = A_1 \delta A_2 \text{ where } A_1 \subseteq A_1 \text{ and } A_2 \subseteq A_2 \text{ and } \delta' \subseteq \delta.
4. If A' \subseteq A_1 \delta A_2 then A' = A_1 \delta' A_2 \text{ where } A_1 \subseteq A_1 \text{ and } A_2 \subseteq A_2 \text{ and } \delta' \subseteq \delta.
5. If A_1 \rightarrow A_2 \subseteq A then A = A_1 \rightarrow A_2 \text{ where } A_1 \subseteq A_1 \text{ and } A_2 \subseteq A_2.
6. If A' \subseteq A_1 \rightarrow A_2 then A' = A_1 \rightarrow A_2 \text{ where } A_1 \subseteq A_1 \text{ and } A_2 \subseteq A_2.

Proof:
1. By case analysis on Unit \( \subseteq A \).
   • Case Unit \( \subseteq \text{Unit} \): Immediate that A = Unit.
2. Symmetric to the previous statement, hence omitted.
3. By case analysis on A', \delta' \subseteq A.
   • Case A', \delta' \subseteq A \text{ where } A = A_1 \delta A_2 \text{ and } \delta' \subseteq \delta: Immediate as A = A_1 \delta A_2 \text{ and subderivations are } A_1 \subseteq A_1 \text{ and } A_2 \subseteq A_2 \text{ and } \delta' \subseteq \delta.
4. Symmetric to the previous statement, hence omitted.
5. By case analysis on A' \subseteq A_2 \subseteq A.
   • Case A_1 \rightarrow A_2 \subseteq A_1 \rightarrow A_2 \text{ where } A = A_1 \rightarrow A_2 \text{ and subderivations are } A_1 \subseteq A_1 \text{ and } A_2 \subseteq A_2.
6. Symmetric to the previous statement, hence omitted.

Lemma 5 (Reflexivity of precision).
For all types A, it is the case that A \( \subseteq A \).

Proof: By induction on the structure of A.

• Case A = Unit: By the definition of precision, A \( \subseteq A \).
• Case A = A_1 \delta A_2: By the induction hypothesis, A_1 \( \subseteq A_1 \text{ and } A_2 \subseteq A_2 \). By the reflexivity of precision on sums, \delta \subseteq \delta. Thus, by the definition of subtyping, A \( \subseteq A \).
• Case A = A_2 \rightarrow A_2: By the induction hypothesis, A_1 \( \subseteq A_1 \text{ and } A_2 \subseteq A_2 \). Thus, by the definition of subtyping, A \( \subseteq A \).

Lemma 6 (Transitivity of precision).
If A_1 \( \subseteq A_2 \text{ and } A_2 \subseteq A_3 \) then A_1 \( \subseteq A_2 \).


• Case A_2 = Unit:
• Case $A_2 = A_{12} \delta_2 A_{22}$:

\[
\begin{align*}
A_1 &\subseteq A_{12} \delta_2 A_{22} \quad \text{Given} \\
A_1 &\subseteq A_{11} \delta_1 A_{21} \quad \text{By Lemma 4 (Precision inversion)} \\
A_{12} &\subseteq A_{13} \quad \text{"} \\
A_{22} &\subseteq A_{23} \quad \text{"} \\
\delta_1 &\subseteq \delta_2 \quad \text{"} \\
A_{12} \delta_2 A_{22} &\subseteq A_3 \quad \text{Given} \\
A_3 &\subseteq A_{13} \delta_3 A_{23} \quad \text{By Lemma 4 (Precision inversion)} \\
A_{12} &\subseteq A_{13} \quad \text{"} \\
A_{22} &\subseteq A_{23} \quad \text{"} \\
\delta_2 &\subseteq \delta_3 \quad \text{"} \\
A_{11} &\subseteq A_{13} \quad \text{By the induction hypothesis} \\
A_{21} &\subseteq A_{23} \quad \text{By the induction hypothesis} \\
\delta_1 &\subseteq \delta_3 \quad \text{By transitivity of } \subseteq \\
A_{11} \delta_1 A_{21} &\subseteq A_{13} \delta_3 A_{23} \quad \text{By the definition of } \subseteq \\
A_1 &\subseteq A_3 \quad \text{Equivalent}
\end{align*}
\]

• Case $A_2 = A_{12} \rightarrow A_{22}$:

\[
\begin{align*}
A_1 &\subseteq A_{12} \rightarrow A_{22} \quad \text{Given} \\
A_1 &\subseteq A_{11} \rightarrow A_{21} \quad \text{By Lemma 4 (Precision inversion)} \\
A_{11} &\subseteq A_{12} \quad \text{"} \\
A_{21} &\subseteq A_{22} \quad \text{"} \\
A_{12} \rightarrow A_{22} &\subseteq A_3 \quad \text{Given} \\
A_3 &\subseteq A_{13} \rightarrow A_{23} \quad \text{By Lemma 4 (Precision inversion)} \\
A_{12} &\subseteq A_{13} \quad \text{"} \\
A_{22} &\subseteq A_{23} \quad \text{"} \\
A_{11} &\subseteq A_{13} \quad \text{By the induction hypothesis} \\
A_{21} &\subseteq A_{23} \quad \text{By the induction hypothesis} \\
A_{11} \rightarrow A_{21} &\subseteq A_{13} \rightarrow A_{23} \quad \text{By the definition of } \subseteq \\
A_1 &\subseteq A_3 \quad \text{Equivalent}
\end{align*}
\]

D.1.3 Directed Consistency

\[\text{Lemma 7 (Reflexivity of directed consistency). For all types } A, \text{ it is the case that } A \sim A.\]

\[\text{Proof. Immediate from Lemma 5 (Reflexivity of precision), Lemma 2 (Reflexivity of subtyping) and rule DirConsU.}\]

\[\text{Lemma 8 (Subtyping obeys directed consistency). If } A \leq B \text{ then } A \sim B.\]

\[\text{Proof. By Lemma 5 (Reflexivity of precision), } A \subseteq A \text{ and } B \subseteq B. \text{ It is given that } A \leq B. \text{ Therefore, by rule DirConsU, } A \sim B.\]

\[\text{Lemma 9 (Loss in precision obeys directed consistency). If } A \subseteq B \text{ then } A \sim B.\]

\[\text{Proof. By Lemma 5 (Reflexivity of precision), } A \subseteq A \text{ and Lemma 2 (Reflexivity of subtyping), } A \leq A. \text{ It is given that } A \subseteq B. \text{ Therefore, by rule DirConsU, } A \sim B.\]

\[\text{Lemma 10 (Gain in precision obeys directed consistency). If } A \subseteq B \text{ then } B \sim A.\]

\[\text{Proof. It is given that } A \subseteq B. \text{ By Lemma 2 (Reflexivity of subtyping), } A \leq A. \text{ By Lemma 5 (Reflexivity of precision), } A \subseteq A. \text{ Therefore, by rule DirConsU, } B \sim A.\]
\[ A' \simeq A \] Type \( A' \) is structurally equivalent to \( A \)

\[
\begin{align*}
\text{Unit} \simeq \text{Unit} \\
A_1' \simeq A_1 & \quad A_2' \simeq A_2 \\
(A_1 \, \delta' \, A_2') \simeq (A_1 \, \delta \, A_2) & \quad (A_1' \rightarrow A_2') \simeq (A_1 \rightarrow A_2)
\end{align*}
\]

Figure 18. Source structural equivalence

D.1.4 Structural Equivalence

Lemma 11 (Reflexivity of Structural Equivalence).
For all types \( A \), it is the case that \( A \simeq A \).

Proof. By induction on the structure of \( A \). All cases are immediate by the induction hypothesis and the definition of \( \simeq \).

Lemma 12 (Symmetry of Structural Equivalence).
If \( A' \simeq A \) then \( A \simeq A' \).

Proof. By structural induction on the derivation of \( A' \simeq A \). All cases are immediate by the induction hypothesis and the definition of \( \simeq \).

Lemma 13 (Transitivity of Structural Equivalence).
If \( A_1 \simeq A_2 \) and \( A_2 \simeq A_3 \) then \( A_1 \simeq A_3 \).

Proof. By induction on the structure of the type \( A_2 \). All cases are immediate from inversion on structural equivalence, the induction hypothesis, and the definition of \( \simeq \).

Corollary 14 (Structural Equivalence is an equivalence relation).
The binary relation \( \simeq \) on types is an equivalence relation.

Proof. Immediate from Lemma 11 (Reflexivity of Structural Equivalence), Lemma 12 (Symmetry of Structural Equivalence), and Lemma 13 (Transitivity of Structural Equivalence).

Lemma 15 (Subtyping obeys Structural Equivalence).
If \( A' \leq A \) then \( A' \simeq A \).

Proof. By induction on the structure of the derivation of \( A' \leq A \).

• Case \( \text{Unit} \leq \text{Unit} \): By definition of structural equivalence, \( \text{Unit} \simeq \text{Unit} \).

• Case \( A_1' \leq A_1 \, A_2' \leq A_2 \, \delta' \leq \delta \)

\[
\frac{(A_1' \, \delta' \, A_2') \leq (A_1 \, \delta \, A_2)}{A_1' \leq A_1 \quad \text{Subderivation}} \quad \frac{A_2' \leq A_2 \quad \text{Subderivation}}{A_1' \simeq A_1 \quad \text{By the induction hypothesis}} \quad \frac{A_2' \simeq A_2 \quad \text{By the induction hypothesis}}{A_1' \, \delta' \, A_2' \simeq A_1 \, \delta \, A_2 \quad \text{By definition of} \, \simeq}
\]

• Case \( A_1 \leq A_1' \, A_2' \leq A_2 \)

\[
\frac{(A_1' \rightarrow A_2') \leq (A_1 \rightarrow A_2)}{A_1 \leq A_1' \quad \text{Subderivation}} \quad \frac{A_2' \leq A_2 \quad \text{Subderivation}}{A_1 \simeq A_1' \quad \text{By the induction hypothesis}} \quad \frac{A_2' \simeq A_2 \quad \text{By Lemma 12 (Symmetry of Structural Equivalence)}}{A_1' \rightarrow A_2' \simeq A_1 \rightarrow A_2 \quad \text{By definition of} \, \simeq}
\]

Lemma 16 (Precision obeys Structural Equivalence).
If \( A' \subseteq A \) then \( A' \simeq A \).

Proof. By induction on the structure of the derivation of \( A' \subseteq A \). All cases are immediate by the induction hypothesis and the definition of structural equivalence.

Lemma 17 (Directed consistency obeys Structural Equivalence).
If \( A \simeq B \) then \( A \simeq B \).
Proof. It is given that $A \simeq B$. By inversion on $\text{DirConsU}$ there exist $A'$ and $B'$ such that $A' \sqsubseteq A$ and $A' \leq B'$ and $B' \sqsubseteq B$. By Lemma 16 (Precision obeys Structural Equivalence), $A' \simeq A$ and $B' \simeq B$. By Lemma 11 (Reflexivity of Structural Equivalence), $A \simeq A'$. By Lemma 13 (Subtyping obeys Structural Equivalence), $A' \simeq B'$. Therefore, by Lemma 19 (Transitivity of Structural Equivalence), $A \simeq B$. 

D.1.5 Decidability

In this section, we write $\mathcal{J}$ decidable in proofs to indicate that the associated judgment form $\mathcal{J}$ is decidable.

$$\delta' \leq \delta$$

Sum $\delta'$ is a sub-sum of $\delta$

<table>
<thead>
<tr>
<th>$+$</th>
<th>$+$</th>
<th>$+$</th>
<th>$+$</th>
<th>$+$</th>
<th>$+$</th>
<th>$+$</th>
<th>$+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
</tr>
</tbody>
</table>

Figure 19. Reflexive, transitive closure of source subsum

Lemma 18 (Decidability of subsum).
Given $\delta'$ and $\delta$, the judgment $\delta' \leq \delta$ is decidable.

Proof. We present the reflexive, transitive closure of the subsum relation on source sums in Figure 19. We can view this relation as a finite set of ordered sums. Thus, the decidability of the subsum relation is equivalent to a membership check on this set.

Lemma 19 (Decidability of subtyping).
Given $A'$ and $A$, the judgment $A' \leq A$ is decidable.

Proceed by case analysis on the head constructors of $A'$ and $A$. Either they agree or they disagree.
If they disagree, then no rule can possibly derive $A' \leq A$.
If they agree, then:

- Case $A' = \text{Unit}$ and $A = \text{Unit}$: By definition of subtyping, $\text{Unit} \leq \text{Unit}$.
- Case $A' = A'_1 \delta A'_2$ and $A = A_1 \delta A_2$:
  - $A'_1 \leq A_1$ decidable: By the induction hypothesis
  - $A'_2 \leq A_2$ decidable: By the induction hypothesis
  - $\delta' \leq \delta$ decidable: By Lemma 18 (Decidability of subsum)
  - $A'_1 \delta' A'_2 \leq A_1 \delta A_2$ decidable: By decidability of premises
- Case $A' = A'_1 \rightarrow A'_2$ and $A = A_1 \rightarrow A_2$:
  - $A_1 \leq A'_1$ decidable: By the induction hypothesis
  - $A'_2 \leq A_2$ decidable: By the induction hypothesis
  - $A'_1 \rightarrow A'_2 \leq A_1 \rightarrow A_2$ decidable: By decidability of premises

$$\delta' \sqsubseteq \delta$$

Sum $\delta'$ is more precise than $\delta$

<table>
<thead>
<tr>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
<th>$\scriptstyle{\times}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
<td>$\scriptstyle{\times}$</td>
</tr>
</tbody>
</table>

Figure 20. Reflexive, transitive closure of precision on sums

Lemma 20 (Decidability of precision on sums).
Given $\delta'$ and $\delta$, the judgment $\delta' \sqsubseteq \delta$ is decidable.

Proof. We present the reflexive, transitive closure of the precision relation on source sums in Figure 20. We could view this relation as a finite set of ordered sums. Thus, the decidability of the precision relation is equivalent to a membership check on this set. Therefore, given $\delta'$ and $\delta$, check whether or not $(\delta', \delta) \in \sqsubseteq$.

Lemma 21 (Decidability of precision on types).
Given $A'$ and $A$, the judgment $A' \sqsubseteq A$ is decidable.

Proceed by case analysis on the head constructors of $A'$ and $A$. Either they agree or they disagree.

If they disagree, then no rule can possibly derive $A' \sqsubseteq A$.

If they agree, then:

- **Case $A' = \text{Unit}$ and $A = \text{Unit}$**: By definition of precision, $\text{Unit} \sqsubseteq \text{Unit}$ and therefore derivability is decidable.

- **Case $A' = A_1' \delta' A_2'$ and $A = A_1 \delta A_2$**:
  
  - $A_1' \sqsubseteq A_1$ decidable: By the induction hypothesis
  
  - $A_2' \sqsubseteq A_2$ decidable: By the induction hypothesis
  
  - $\delta' \sqsubseteq \delta$ decidable: By Lemma 20 (Decidability of precision on sums)
  
- **Case $A' = A_1' \rightarrow A_2'$ and $A = A_1 \rightarrow A_2$**:
  
  - $A_1' \sqsubseteq A_1$ decidable: By the induction hypothesis
  
  - $A_2' \sqsubseteq A_2$ decidable: By the induction hypothesis

Lemma 22 (Decidability of directed consistency).

Given $A'$ and $B'$, the relation $A' \leadsto B'$ is decidable.

Proof. We have $A' \leadsto B'$ if and only if there exist $A$ and $B$ such that $A \sqsubseteq A'$ and $A \leq B$ and $B \sqsubseteq B'$. We are given $A'$; there are only finitely many types such that $A \sqsubseteq A'$. Each such $A$ has only finitely many supertypes, that is, types $B$ such that $A \leq B$. Since these two relations are decidable, $A' \leadsto B'$ is decidable.

Theorem 1 (Decidability of bidirectional typing).

1. Given $\Gamma$, $e$ and $A$, the judgment $\Gamma \vdash e \equiv A$ is decidable.

2. Given $\Gamma$ and $e$, the judgment $\Gamma \vdash e \Rightarrow A$ is decidable.

Proof. By lexicographic induction on (1) the expression $e$, then on (2) the judgment form, with $\Rightarrow$ smaller than $\equiv$.

In most rules, the expression gets smaller in all the premises: $\text{Synt} \rightarrow \text{Syn} \rightarrow \text{Chk}$, $\text{Syn} \rightarrow \text{Elim}$, $\text{ChkIntro}$, $\text{ChkIntro}$, $\text{ChkElim1}$ and $\text{ChkElim2}$.

In $\text{ChkSub}$, the premise types the same expression but is a synthesizing judgment, which is smaller under our induction measure. By Lemma 22, the second premise of $\text{ChkSub}$ is decidable.

D.1.6 Equivalence of type assignment and bidirectional system

Lemma 23 (All sums below $\times$).

For all source sums $\delta$, it is the case that $\delta \leq \times$.

Proof. By case analysis on $\delta$.

- **Case $\delta = +^*$**: By the definition of subtyping, $+^* \leq \times$.

- **Case $\delta = +_i$**: By the definition of subtyping, $+_i \leq +^*_i$. By the previous case, $+^*_i \leq \times$. By the transitivity of subtyping, $+_i \leq \times$.

- **Case $\delta = +_i$**: By the definition of subtyping, $+^*_i \leq +_i$. By the previous case, $+_i \leq \times$. By the transitivity of subtyping, $+^*_i \leq \times$.

- **Case $\delta = +_i^*$**: By the definition of subtyping, $+_i \leq +_i^*$. By the definition of subtyping, $+_i \leq \times$. By the transitivity of subtyping, $+_i^* \leq \times$.

Lemma 24 ($\Rightarrow$ implies subsum).

If $\delta' \Rightarrow \delta$ then $\delta' \leq \delta$.

Proof. By case analysis on $\delta' \Rightarrow \delta$.

- **Case $+_i \Rightarrow +_i^*$**: By definition of subtyping, $+_i \leq +_i^*$. By definition of subtyping, $+_i \leq +^*_i$. By transitivity of subtyping, $+^*_i \leq +^*_i$.

- **Case $+_i \Rightarrow +^*_i$**: By definition of subtyping, $+_i \leq +^*_i$.

- **Case $+_i \Rightarrow +^*_i$**: By definition of subtyping, $+_i \leq +^*_i$.

- **Case $+_i \Rightarrow +^*_i$**: By reflexivity of subtyping, $+_i \leq +^*_i$.

- **Case $\delta' \Rightarrow \times$**: By Lemma 23 (All sums below $\times$), $\delta' \leq \times$.

Theorem 2 (Bidirectional soundness).

If $\Gamma \vdash e \equiv A$ or $\Gamma \vdash e \Rightarrow A$ then $\Gamma \vdash e : A$.

Proof. By induction on the structure of the given derivation.

- **Case $\text{SyntVar}$**: Apply rule $\text{SVar}$

- **Case $\text{ChkSub}$**: Use the induction hypothesis and apply rule $\text{SCSub}$
• Case \textbf{SynAnno} Use the induction hypothesis, and apply rule\textbf{SAnno}.

• Case \textbf{ChkUnitIntro} Apply rule\textbf{SUnitIntro}.

• Case

\[
\begin{align*}
\Gamma \vdash e_0 &\in A_1 & \ast_1 \leq \delta & \text{Case 1} \\
\Gamma \vdash inj_i e_0 &\in (A_1 \delta A_2) & & \text{ChkSumIntro}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_0 &\in A_1 & \text{Subderivation} \\
\Gamma \vdash e_0 : A_1 & & \text{By the induction hypothesis} \\
\Gamma \vdash inj_i e_0 : (A_1 \ast_1 A_2) & & \text{By rule}\textbf{SSumIntro} \\
A_1 \leq A_1 & & \text{By Lemma 2 (Reflexivity of subtyping)} \\
A_2 \leq A_2 & & \text{By Lemma 2 (Reflexivity of subtyping)} \\
\ast_1 \leq \delta & & \text{Subderivation} \\
A_1 \ast_1 A_2 \leq A_1 \delta A_2 & & \text{By definition of \leq} \\
A_1 \ast_1 A_2 \sim A_1 \delta A_2 & & \text{By Lemma 8 (Subtyping obeys directed consistency)} \\
\Gamma \vdash inj_i e_0 : (A_1 \delta A_2) & & \text{By rule}\textbf{SCSub}
\end{align*}
\]

• Case

\[
\begin{align*}
\Gamma \vdash e_0 &\in (A_1 \delta A_2) & \delta \Rightarrow \ast_1 & \text{Case 2} \\
\Gamma \vdash \text{case}(e_0, inj_i x.e_i) &\in A & \text{ChkSumElim1}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_0 &\Rightarrow (A_1 \delta A_2) & \text{Subderivation} \\
\Gamma \vdash e_0 : (A_1 \delta A_2) & & \text{By the induction hypothesis} \\
\delta \Rightarrow \ast_1 & & \text{Subderivation} \\
A_1 \leq A_1 & & \text{By Lemma 2 (Reflexivity of subtyping)} \\
A_2 \leq A_2 & & \text{By Lemma 2 (Reflexivity of subtyping)} \\
\ast_1 \leq \delta & & \text{By Lemma 2 (Reflexivity of subtyping)} \\
A_1 \delta A_2 \leq A_1 \ast_1 A_2 & & \text{By definition of \leq} \\
A_1 \delta A_2 \sim A_1 \ast_1 A_2 & & \text{By Lemma 8 (Subtyping obeys directed consistency)} \\
\Gamma \vdash e_0 : (A_1 \ast_1 A_2) & & \text{By rule}\textbf{SCSub} \\
\Gamma, x : A_1 \vdash e_i \in A & & \text{Subderivation} \\
\Gamma, x : A_1 \vdash e_i : A & & \text{By the induction hypothesis} \\
\Gamma \vdash \text{case}(e_0, inj_i x.e_i) : A & & \text{By rule}\textbf{ChkSumElim1}
\end{align*}
\]

• Case \textbf{ChkSumElim2} Similar to the\textbf{ChkSumElim1} case, hence omitted.

• Case \textbf{ChkSumIntro} Use the induction hypothesis, and apply rule\textbf{SIntro}.

• Case \textbf{SynElim} Use the induction hypothesis, and apply rule\textbf{SElim}.

\begin{proof}
By induction on the structure of \(e\).
All cases either hold directly by definition or by first using the induction hypothesis.
\end{proof}

\begin{lemma}[Reflexivity of annotation equivalence] For all expressions \(e, e \equiv e\).
\end{lemma}

\begin{proof}
By induction on the structure of \(e\).
\end{proof}

\begin{lemma}[Synthesis also checks] If \(\Gamma \vdash e \Rightarrow A\) then \(\Gamma \vdash e \Leftarrow A\).
\end{lemma}

\begin{proof}
Apply rule\textbf{ChkCSub} as \(A \sim A\) holds by Lemma 5 (Reflexivity of precision).
\end{proof}

\begin{theorem}[Annotatability]
If \(\Gamma \vdash e : A\) then there exist \(e^\prime\) and \(e^\prime\) such that (1) \(\Gamma \vdash e^\prime \Leftarrow A\) where \(e \equiv e^\prime\), and (2) \(\Gamma \vdash e^\prime \Rightarrow A\) where \(e \equiv e^\prime\).
\end{theorem}

\begin{proof}
By induction on the structure of the derivation of \(\Gamma \vdash e : A\).
\end{proof}

• Case \(\Gamma(x) = A\)

\[
\begin{align*}
\Gamma \vdash x : A & & \text{SVar}
\end{align*}
\]

\[
\begin{align*}
\Gamma(x) = A & & \text{Premise} \\
\Rightarrow \Gamma \vdash x : A & & \text{By rule}\textbf{SynVar} \\
\Rightarrow \Gamma \vdash x \Leftarrow A & & \text{By Lemma 26 (Synthesis also checks)} \\
\Rightarrow x \equiv x & & \text{By definition of \equiv}
\end{align*}
\]
• **Case** \[ \Gamma \vdash e : A' \Rightarrow A' \Rightarrow A \]
  
  \[
  \begin{array}{c}
  \Gamma \vdash e : A' \\
  \Gamma \vdash e' : A' \Rightarrow A' \\
  e =: e'
  \end{array}
  \]

  Subderivation

  By the induction hypothesis

  By definition of =:

• **Case** \[ \Gamma \vdash e_0 : A \]

  \[
  \begin{array}{c}
  \Gamma \vdash (e_0 :: A) : A \\
  \Gamma \vdash e_0 : A \Rightarrow A \\
  \Gamma \vdash e_0' \Rightarrow A \\
  e_0 =: e_0'
  \end{array}
  \]

  Subderivation

  By rule **ChkCSub**

  By rule **SynAnno**

  By Lemma [26] (Synthesis also checks)

  By definition of =:

• **Case** \[ \Gamma \vdash () : Unit \]

  \[
  \begin{array}{c}
  \Gamma \vdash () \leftarrow Unit \Rightarrow Unit \\
  \Gamma \vdash () \Rightarrow Unit \Rightarrow Unit \\
  () =: () \\
  () =: () : Unit \Rightarrow Unit \\
  \end{array}
  \]

  By rule **ChkUnitIntro**

  By rule **SynAnno**

  By definition of =:

• **Case** \[ \Gamma \vdash e_0 : A_1 \]

  \[
  \begin{array}{c}
  \Gamma \vdash inj_i e_0 : (A_1 : A_2) \\
  \Gamma \vdash e_0 : A_1 \\
  \Gamma \vdash e_0' \leftarrow A_1 \\
  e_0 =: e_0'
  \end{array}
  \]

  Subderivation

  By the induction hypothesis

  By definition of \\

  By definition of \\

• **Case** \[ \Gamma \vdash e_0 : A_1 +_1 A_2 \]

  \[
  \begin{array}{c}
  \Gamma \vdash case(e_0, inj_{i} x.e_i) : A \\
  \Gamma \vdash e_0 : A_1 +_1 A_2 \\
  \Gamma \vdash e_0' \Rightarrow A_1 +_1 A_2 \\
  e_0 =: e_0'
  \end{array}
  \]

  Subderivation

  By the induction hypothesis

  By definition of \\

  By definition of \\

• **Case** \[ \Gamma, x : A_1 \vdash e_1 : A \]

  \[
  \begin{array}{c}
  \Gamma \vdash case(e_0, inj_{i} x.e_i) : A \\
  \Gamma \vdash case(e_0, inj_{i} x.e_i) \leftarrow A \\
  \Gamma \vdash case(e_0, inj_{i} x.e_i) \Rightarrow A \\
  \end{array}
  \]

  By rule **ChkSumElim1**

  By rule **ChkSumIntro**

  By rule **SynAnno**

  By definition of =:

  By definition of =:

• **Case** \[ \Gamma, x : A_1 \vdash e_0 : A_2 \]

  \[
  \begin{array}{c}
  \Gamma \vdash case(e_0, inj_{i} x.e_i) : A \\
  \Gamma \vdash case(e_0, inj_{i} x.e_i) \leftarrow A \\
  \Gamma \vdash case(e_0, inj_{i} x.e_i) \Rightarrow A \\
  \end{array}
  \]

  By rule **ChkSumElim2**

  Similar to the **SSumElim1** case, hence omitted.
\[\begin{align*}
& \Gamma, x : A_1 \vdash e_0 : A_2 \\
& \Gamma, x : A_1 \vdash e_0' \iff A_2 \\
& e_0 \equiv e_0' \tag*{Subderivation} \\
& \therefore \quad \Gamma \vdash \lambda x. e_0' \equiv (A_1 \rightarrow A_2) \quad \text{By rule ChkIntro} \\
& \therefore \quad \Gamma \vdash (\lambda x. e_0') : A_1 \rightarrow (A_1 \rightarrow A_2) \quad \text{By rule SynAnno} \\
& \therefore \quad \lambda x. e_0 \equiv \lambda x. e_0' \quad \text{By definition of \( \equiv \)} \\
& \therefore \quad \lambda x. e_0 \equiv (\lambda x. e_0' : A_1 \rightarrow A_2) \\
& \therefore \quad \lambda x. e_0 \equiv (\lambda x. e_0' : A_1 \rightarrow A_2) \quad \text{By definition of \( \equiv \)}
\end{align*}\]

**Case** \( \Gamma \vdash e_1 : A_1 \rightarrow A_2 \quad \Gamma \vdash e_2 : A_1 \)

\[\begin{align*}
& \Gamma \vdash e_1 : A_1 \rightarrow A_2 \\
& \Gamma \vdash e_1' \Rightarrow A_1 \rightarrow A_2 \\
& e_1 \equiv e_1' \tag*{Subderivation} \\
& \Gamma \vdash e_2 : A_1 \\
& \Gamma \vdash e_2' \Leftarrow A_1 \\
& e_2 \equiv e_2' \tag*{Subderivation} \\
& \therefore \quad \Gamma \vdash e_1' \equiv e_2' \quad \text{By rule SynElim} \\
& \therefore \quad \Gamma \vdash e_1' \equiv e_2' \quad \text{By Lemma 26 (Synthesis also checks)} \\
& \therefore \quad e_1 \equiv e_2 \quad \text{By definition of \( \equiv \)}
\end{align*}\]

\[\square\]

**D.2 Typability under varying precision**

**Lemma 27** (Pointwise precision preserves domain).
\[\text{If } \Gamma' \subseteq \Gamma \text{ then } \text{dom}(\Gamma') = \text{dom}(\Gamma).\]

*Proof.* By induction on the structure of \( \Gamma' \subseteq \Gamma \). \[\square\]

**Lemma 28** (Context strengthening).
\[\text{If } \Gamma, y : A' \vdash e : A_0 \text{ and } A' \subseteq A' \text{ then } \Gamma, y : A \vdash e : A_0.\]

*Proof.* By induction on the structure of the derivation of \( \Gamma, y : A' \vdash e : A_0 \).

**Case** \( (\Gamma, y : A')(e) = A_0 \)

\[\begin{align*}
& \Gamma, y : A' \vdash e : A_0 \\
& \therefore \quad \text{Either } e = y, \text{ or } e \neq y. \\
& \text{In the first case:} \\
& (\Gamma, y : A')(y) = A_0 \\
& A' = A_0 \\
& \Gamma, y : A \vdash y : A \\
& A \subseteq A' \\
& A \sim A' \\
& \Gamma, y : A \vdash y : A' \\
& \Gamma, y : A \vdash e : A_0 \\
& \text{By above equalities}
\end{align*}\]

**Case** \( (\Gamma, y : A')(e) = A_0 \)

\[\begin{align*}
& \Gamma, y : A \vdash e : A_0 \\
& \therefore \quad \text{Use the induction hypothesis and apply rule SVar} \\
& \text{Case SUnitIntro} \quad \text{Immediate from rule SUnitIntro} \\
& \text{Case SSumIntro} \quad \text{Use the induction hypothesis and apply rule SSumIntro} \\
& \text{Case SSumElim1} \quad \text{Use the induction hypothesis and apply rule SSumElim1} \\
& \text{Case SSumElim2} \quad \text{Use the induction hypothesis and apply rule SSumElim2} \\
& \text{Case SIntro} \quad \text{Use the induction hypothesis and apply rule SIntro} \\
& \text{Case SElim} \quad \text{Use the induction hypothesis and apply rule SElim}
\end{align*}\]

**Corollary 29.**
\[\text{If } \Gamma' \vdash e : A \text{ and } \Gamma \subseteq \Gamma' \text{ then } \Gamma \vdash e : A.\]
Proof. By induction on the number of variables $x$ such that $x \in \text{dom}(\Gamma')$ but $\Gamma'(x) \neq \Gamma(x)$.
Note that we don’t impose $x \in \text{dom}(\Gamma)$ as $\text{dom}(\Gamma') = \text{dom}(\Gamma')$ by Lemma 27 (Pointwise precision preserves domain).
If $\Gamma'(x) = \Gamma(x)$ for all $x \in \text{dom}(\Gamma')$, then $\Gamma = \Gamma'$ so we already have the result.
Otherwise, use the induction hypothesis, and apply Lemma 28 (Context strengthening).

Lemma 30 (Relating $+_1$-subsum and precision).
If $+_i^1 \leq \delta'$ and $\delta' \sqsubseteq \delta$ then $+_i^1 \leq \delta$.

Proof. Proceed by case analysis on $+_i^1 \leq \delta'$.

- Case $+_i^1 \leq +_i^1$: From the definition of precision, either $\delta = +_i^1$ or $\delta = +^2$. In both cases, there exists a derivation for $+_i^1 \leq \delta$.
- Case $+_i^1 \leq +_1$: From the definition of precision, either $\delta = +_1$, $\delta = +_i^1$, $\delta = +^2$ or $\delta = +^2$. In all cases, there exists a derivation for $+_i^1 \leq \delta$.
- Case $+_i^1 \leq +^2$: From the definition of precision, $\delta = +^2$. We are given a derivation for $+_i^1 \leq +^2$.
- Case $+_i^1 \leq +^2$: From the definition of precision, either $\delta = +^*, \delta = +^2$. In both cases, there exists a derivation for $+_i^1 \leq \delta$.
- Case $+_i^1 \leq +$: From the definition of precision, either $\delta = +$ or $\delta = +^2$. In both cases, there exists a derivation for $+_i^1 \leq \delta$.

Lemma 31 (Bidirectional sum precision).
If $\delta' \Rightarrow \delta_1$ and $\delta' \sqsubseteq \delta$ then $\delta \Rightarrow \delta_1$.

Proof. Proceed by case analysis on $\delta' \Rightarrow \delta_1$.

- Case $+_i^1 \Rightarrow +_i^1$: From the definition of precision, either $\delta = +_i^1$ or $\delta = +^2$. In both cases, there exists a derivation for $\delta \Rightarrow +_i^1$.
- Case $+_i \Rightarrow +_i^1$: From the definition of precision, either $\delta = +_i$, $\delta = +_i^1$, $\delta = +^2$, $\delta = +^2$. In all cases, there exists a derivation for $\delta \Rightarrow +_i^1$.
- Case $+_i \Rightarrow +^2$: From the definition of precision, $\delta = +^2$. We are given a derivation for $+_i \Rightarrow +_i^1$.
- Case $+_i \Rightarrow +^2$: From the definition of precision, either $\delta = +^*, \delta = +^2$. In both cases, there exists a derivation for $\delta \Rightarrow +_i^1$.
- Case $\delta' \Rightarrow +$: There exists a derivation for $\delta \Rightarrow +$ for all $\delta$.

Theorem 4 (Varying precision of bidirectional typing).
1. If $\Gamma' \vdash e' \iff A'$ and $e' \sqsubseteq e$ and $\Gamma' \sqsubseteq \Gamma$ and $A' \sqsubseteq A$
then $\Gamma \vdash e \iff A$.
2. If $\Gamma' \vdash e' \Rightarrow A'$ and $e' \sqsubseteq e$ and $\Gamma' \sqsubseteq \Gamma$
then there exists $A$ such that $\Gamma \vdash e \Rightarrow A$ and $A' \sqsubseteq A$.

Proof. By induction on the structure of the given derivation.

1. By case analysis on the rule concluding $\Gamma' \vdash e' \iff A'$.

- Case

\[
\frac{\Gamma' \vdash \text{ChkUnitIntro}}{
\Gamma' \vdash \text{UnitIntro} \ \ A'}
\]

\[
\text{ChkUnitIntro}
\]

\[
\begin{align*}
( & \subseteq e \quad \text{Given} \\
& e \equiv () \quad \text{From definition of } \subseteq
\end{align*}
\]

\[
\begin{align*}
\text{Unit} \subseteq A & \quad \text{Given} \\
A = \text{Unit} & \quad \text{By Lemma } 4 \text{ (Precision inversion)}
\end{align*}
\]

\[
\Gamma \vdash e \iff \text{Unit} \quad \text{By rule } \text{ChkUnitIntro}
\]

- Case

\[
\frac{\Gamma' \vdash A'_a \quad A'_a \bowtie A'}{
\Gamma' \vdash e' \iff A'}
\]

\[
\text{ChkCSub}
\]

2020/8/16
\[\Gamma \vdash e' \Rightarrow A_0' \quad \text{Subderivation}\]

\[e' \sqsubseteq e \quad \text{Given}\]

\[\Gamma' \sqsubseteq \Gamma \quad \text{Given}\]

\[\Gamma' \vdash e \Rightarrow A_0 \quad \text{By the induction hypothesis}\]

\[A_0' \sqsubseteq A_0' \quad \text{''}\]

\[A_0' \sim A' \quad \text{Subderivation}\]

\[B_0 \sqsubseteq A_0' \quad \text{By inversion on DirConsU}\]

\[B_0 \leq B' \quad \text{''}\]

\[B' \sqsubseteq A' \quad \text{''}\]

\[B_0 \sqsubseteq A_0 \quad \text{By Lemma 6 (Transitivity of precision)}\]

\[A' \quad \text{Given}\]

\[B' \sqsubseteq A \quad \text{By Lemma 6 (Transitivity of precision)}\]

\[A_0 \quad \text{By rule DirConsU}\]

\[\Gamma \vdash e \Leftrightarrow A \quad \text{By rule ChkCSub}\]

\[\text{• Case } \Gamma', x : A_1' \vdash e_0' \Leftrightarrow A_2' \quad \text{ChkIntro}\]

\[\lambda x. e_0' \sqsubseteq e \quad \text{Given}\]

\[e = \lambda x. e_0 \quad \text{From definition of } \sqsubseteq\]

\[e_0' \sqsubseteq e_0 \quad \text{''}\]

\[A_1' \rightarrow A_2' \sqsubseteq A \quad \text{By Lemma 4 (Precision inversion)}\]

\[A = A_1 \rightarrow A_2\]

\[A_1' \sqsubseteq A_1 \quad \text{''}\]

\[A_2' \sqsubseteq A_2 \quad \text{''}\]

\[\Gamma' \sqsubseteq \Gamma \quad \text{Given}\]

\[\Gamma', x : A_1' \sqsubseteq \Gamma, x : A_1 \quad \text{By definition of } \sqsubseteq\]

\[\Gamma', x : A_1' \vdash e_0' \Leftrightarrow A_2' \quad \text{Subderivation}\]

\[\Gamma, x : A_1' \vdash e_0 \Leftrightarrow A_2 \quad \text{By the induction hypothesis}\]

\[\Gamma \vdash \lambda x. e_0 \Leftrightarrow A_1 \rightarrow A_2 \quad \text{By rule ChkIntro}\]

\[\text{• Case } \Gamma' \vdash e_0' \Leftrightarrow A_1' \quad +_1 \leq \delta' \quad \text{ChkSumIntro}\]

\[\text{inj}_1 e_0' \sqsubseteq e \quad \text{Given}\]

\[e = \text{inj}_1 e_0 \quad \text{From definition of } \sqsubseteq\]

\[e_0' \sqsubseteq e_0 \quad \text{''}\]

\[A_1' \delta' A_2' \sqsubseteq A \quad \text{Given}\]

\[A = A_1 \delta A_2\]

\[A_1' \sqsubseteq A_1 \quad \text{''}\]

\[A_2' \sqsubseteq A_2 \quad \text{''}\]

\[\Gamma' \vdash e_0' \Leftrightarrow A_1' \quad \text{Subderivation}\]

\[\Gamma' \sqsubseteq \Gamma \quad \text{Given}\]

\[\Gamma \vdash e_0 \Leftrightarrow A_1 \quad \text{By the induction hypothesis}\]

\[+_1 \leq \delta' \quad \text{Subderivation}\]

\[+_1 \leq \delta \quad \text{By Lemma 30 (Relating +_1-subsum and precision)}\]

\[\Gamma \vdash \text{inj}_1 e_0' \Leftrightarrow (A_1 \delta A_2) \quad \text{By rule ChkSumIntro}\]

\[\text{• Case } \Gamma' \vdash e_0' \Rightarrow A_1' \delta' A_2' \quad \text{ChkSumElim1}\]

\[\delta' \Rightarrow +_1' \quad \Gamma', x : A_1' \vdash e_0' \Leftrightarrow A' \]

\[\Gamma' \vdash \text{case}(e_0', \text{inj}_1 x, e_1') \Leftrightarrow A' \]
\[ e' \subseteq e \quad \text{Given} \]
\[ e = \text{case}(e_0, \text{inj}_1 x, e_1) \quad \text{From definition of \( \subseteq \)} \]
\[ e'_0 \subseteq e_0 \quad \text{"} \]
\[ e'_1 \subseteq e_1 \quad \text{"} \]
\[ \Gamma \vdash e'_0 \Rightarrow A'_1 \delta' A'_2 \quad \text{Subderivation} \]
\[ \Gamma' \subseteq \Gamma \quad \text{Given} \]
\[ \Gamma \vdash e_0 \Rightarrow A_1 \delta A_2 \quad \text{By the induction hypothesis} \]
\[ A'_1 \delta' A'_2 \subseteq A_1 \delta A_2 \quad \text{"} \]
\[ A'_1 \subseteq A_1 \quad \text{From definition of \( \subseteq \)} \]
\[ A'_2 \subseteq A_2 \quad \text{"} \]
\[ \delta' \Rightarrow +' \quad \text{Subderivation} \]
\[ \delta \Rightarrow +' \quad \text{By Lemma 31 (Bidirectional sum precision)} \]
\[ \Gamma', x : A'_1 \subseteq \Gamma, x : A_1 \quad \text{By definition of \( \subseteq \)} \]
\[ A' \subseteq A \quad \text{Given} \]
\[ \Gamma', x : A'_1 \vdash e'_1 \Leftrightarrow A' \quad \text{Subderivation} \]
\[ \Gamma, x : A_1 \vdash e_1 \Leftrightarrow A \quad \text{By the induction hypothesis} \]
\[ \Gamma \vdash \text{case}(e_0, \text{inj}_1 x, e_1) \Leftrightarrow A' \quad \text{By rule ChkSumElim1} \]

- **Case**
  \[ \Gamma' \vdash e_0 \Rightarrow A'_1 \delta' A'_2 \quad \text{Subderivation} \]
  \[ \delta \Rightarrow +' \quad \text{By Lemma 31 (Bidirectional sum precision)} \]
  \[ \Gamma' \vdash e_0 \Rightarrow A_1 \delta A_2 \quad \text{By the induction hypothesis} \]
  \[ A'_1 \delta' A'_2 \subseteq A_1 \delta A_2 \quad \text{"} \]
  \[ A'_1 \subseteq A_1 \quad \text{From definition of \( \subseteq \)} \]
  \[ A'_2 \subseteq A_2 \quad \text{"} \]
  \[ \delta' \Rightarrow +' \quad \text{Subderivation} \]
  \[ \delta \Rightarrow +' \quad \text{By Lemma 31 (Bidirectional sum precision)} \]
  \[ A' \subseteq A \quad \text{Given} \]

\[ \Gamma', x_1 : A'_1 \subseteq \Gamma, x_1 : A_1 \quad \text{By definition of \( \subseteq \)} \]
\[ \Gamma', x_1 : A'_1 \vdash e'_1 \Leftrightarrow A' \quad \text{Subderivation} \]
\[ \Gamma, x_1 : A_1 \vdash e_1 \Leftrightarrow A \quad \text{By the induction hypothesis} \]
\[ \Gamma', x_2 : A'_2 \subseteq \Gamma, x_2 : A_2 \quad \text{By definition of \( \subseteq \)} \]
\[ \Gamma', x_2 : A'_2 \vdash e'_2 \Leftrightarrow A' \quad \text{Subderivation} \]
\[ \Gamma, x_2 : A_2 \vdash e_2 \Leftrightarrow A \quad \text{By the induction hypothesis} \]
\[ \Gamma \vdash \text{case}(e_0, \text{inj}_1 x_1, e_1, \text{inj}_2 x_2, e_2) \Leftrightarrow A \quad \text{By rule ChkSumElim2} \]

2. By case analysis on the rule concluding \( \Gamma' \vdash e' \Rightarrow A' \).

- **Case**
  \[ \Gamma'(x) = A' \quad \text{SynVar} \]
  \[ \Gamma' \vdash \underbrace{x}_{	ext{e'}} \Rightarrow A' \quad \text{SynVar} \]

Let \( A = \Gamma(x) \).
\[ x \sqsubseteq e \quad \text{Given} \\
\text{e} = x \quad \text{From definition of } \sqsubseteq \\
\Gamma'(x) = A' \quad \text{Premise} \\
\Gamma' \sqsubseteq \Gamma \quad \text{Given} \\
\Gamma'(x) \sqsubseteq \Gamma(x) \quad \text{By definition of } \sqsubseteq \text{ on contexts} \\
\forall A' \sqsubseteq A \quad \text{Equivalent} \\
\Gamma \vdash x \Rightarrow A \quad \text{By rule } \text{SynVar} \\
\]  

- **Case**  
  \[ \Gamma' \vdash e'_0 \iff A' \]
  \[ \Gamma' \vdash (e'_0 : A') \Rightarrow A \quad \text{SynAnno} \]
  \[ (e'_0 : A') \sqsubseteq e \quad \text{Given} \\
  \text{e} = (e_0 : A_0) \quad \text{From definition of } \sqsubseteq \\
  e'_0 \sqsubseteq e_0 \quad " \\
\]  

- **Case**  
  \[ \Gamma' \vdash e'_1 \Rightarrow A'_0 \rightarrow A' \]
  \[ \Gamma' \vdash e'_2 \iff A'_0 \]
  \[ \Gamma' \vdash e'_1, e'_2 \Rightarrow A' \quad \text{Syn→Elim} \]
  \[ e'_1, e'_2 \sqsubseteq e \quad \text{Given} \\
  \text{e} = e_1, e_2 \quad \text{From definition of } \sqsubseteq \\
  e'_1 \sqsubseteq e_1 \quad " \\
  e'_2 \sqsubseteq e_2 \quad " \\
\]  

\[ \text{Lemma 32 (Static looseness).} \]
\[ \text{If } +_1^1 \leq \delta^5 \text{ then } +_1 \leq_5 \delta^5. \]

**Proof.** By case analysis on \( +_1^1 \leq \delta^5 \).

- **Case** \( +_1^1 \leq +_1^2 \) : By definition of static subtyping \( +_1 \leq_5 +_1^2 \).

- **Case** \( +_1^1 \leq +^* \) : By definition of static subtyping \( +_1 \leq_5 +^* \).

\[ \text{Lemma 33 (Static looseness, II).} \]
\[ \text{If } \delta^5 \Rightarrow +_1^* \text{ then } \delta^5 = +_1^*. \]

**Proof.** By case analysis on \( \delta^5 \Rightarrow +_1^* \).

- **Case** \( +_1 \Rightarrow +_1^* \) : It is the case that \( \delta^5 = +_1 \).

The following lemma states that static sums are the most precise and incomparable by the precision relation.

\[ \text{Lemma 34 (Precision for static sums).} \]
\[ \text{If } \delta_1 \sqsubseteq \delta^5_2 \text{ then } \delta_1 = \delta^5_2. \]

**Proof.** Proceed by case analysis on \( \delta^5_2 \).

\[ \text{D.3 Properties of the Static System} \]

\[ \text{Lemma 32 (Static looseness).} \]
\[ \text{If } +_1^1 \leq \delta^5 \text{ then } +_1 \leq_5 \delta^5. \]

**Proof.** By case analysis on \( +_1^1 \leq \delta^5 \).

- **Case** \( +_1^1 \leq +_1^2 \) : By definition of static subtyping \( +_1 \leq_5 +_1^2 \).

- **Case** \( +_1^1 \leq +^* \) : By definition of static subtyping \( +_1 \leq_5 +^* \).

\[ \text{Lemma 33 (Static looseness, II).} \]
\[ \text{If } \delta^5 \Rightarrow +_1^* \text{ then } \delta^5 = +_1^*. \]

**Proof.** By case analysis on \( \delta^5 \Rightarrow +_1^* \).

- **Case** \( +_1 \Rightarrow +_1^* \) : It is the case that \( \delta^5 = +_1 \).
Proof.

Lemma 37 (Equivalence for static subtyping).
If \( A_1 \subseteq A_2 \) then \( A_1 = A_2 \).

Proof. By induction on the structure of \( A_2 \).

1. **Case \( A_2 = \text{Unit} \):** By the definition of imprecision, \( A_1 = \text{Unit} \) only.

2. **Case \( A_2 = A_3 A_4 : \)**
   - Given \( A_1 \subseteq A_5 \delta_2 A_5 \delta_2 A_5 \):
     - \( A_1 = A_1 \delta_1 A_2 \) from the definition of \( \subseteq \).
     - \( A_1 \delta_2 A_2 \) and \( \delta_1 \subseteq \delta_2 \) by the induction hypothesis.
     - \( A_1 = A_2 \delta_2 A_5 \) by Definition of \( = \).
     - Given \( \delta_1 \delta_2 \) by Lemma 34 (Precision for static sums).

3. **Case \( A_2 = A_5 \):** Similar to the previous case.

Lemma 36 (Equivalence for static sums).
1. If \( \delta_1 \leq \delta_2 \) then \( \delta_1 \leq \delta_2 \).
2. If \( \delta_1 \leq \delta_2 \) then \( \delta_1 \leq \delta_2 \).

Proof.
1. By case analysis on \( \delta_1 \leq \delta_2 \).
   - **Case \( \delta_1 \leq \delta_2 \):** By definition of subtyping, \( \delta_1 \leq \delta_2 \).
   - **Case \( +_1 \leq +_2 \):** By definition of subtyping, \( +_1 \leq +_2 \).

2. By case analysis on \( \delta_1 \leq \delta_2 \).
   - **Case \( \delta_1 \leq \delta_2 \):** By definition of static subtyping, \( \delta_1 \leq \delta_2 \).
   - **Case \( +_1 \leq +_2 \):** By definition of static subtyping, \( +_1 \leq +_2 \).

Lemma 37 (Equivalence for static subtyping).
1. If \( A_1 \delta_1 \leq A_2 \) then \( A_1 \leq A_2 \).
2. If \( A_1 \delta_1 \leq A_2 \) then \( A_1 \leq A_2 \).

Proof.
1. By induction on the structure of the derivation of \( A_1 \leq A_2 \).
   - **Case Unit \( \leq \text{Unit} \):** By definition of subtyping, Unit \( \leq \text{Unit} \).

2. **Case \( A_1 \leq A_2 \):**
   - **Case \( A_1 \delta_1 \leq A_2 \delta_2 \):** By definition of subtyping, \( \delta_1 \leq \delta_2 \).
   - **Case \( A_1 \delta_1 \leq A_2 \delta_2 \):** By definition of subtyping, \( \delta_1 \leq \delta_2 \).

Similar to the previous case.

2. By induction on the structure of the derivation of \( A_1 \delta_1 \leq A_2 \delta_2 \).
   - **Case Unit \( \leq \text{Unit} \):** By definition of subtyping, Unit \( \leq \text{Unit} \).
Proof. Theorem 5 (Static soundness and completeness)

It is given that $A_1^S \leq A_2^S$, $A_3^S \leq A_4^S$, $\delta_1^S \leq \delta_2^S$.

$(A_1^S \delta_1^S A_2^S) \leq (A_2^S \delta_2^S A_3^S)$

$A_1^S \leq A_1^S$ Subderivation
$A_3^S \leq A_2^S$ Subderivation
$\delta_1^S \leq \delta_2^S$ Subderivation
$A_3^S \leq S A_4^S$ By the induction hypothesis
$A_2^S \leq A_3^S$ Subderivation
$\delta_2^S \leq \delta_2^S$ By Lemma 30 (Equivalence for static subsum)
$A_1^S, \delta_1^S A_2^S \leq S A_2^S, \delta_2^S A_3^S$ By definition of $\leq_S$

Case $A_5^S \leq A_1^S, A_3^S \leq A_3^S$
$(A_1^S \rightarrow A_3^S) \leq (A_3^S \rightarrow A_3^S)$

Similar to the previous case. $
$

Lemma 38 (Directed consistency for static types).
If $A_1^S \sim A_2^S$ then $A_1^S \leq A_2^S$.

Proof. It is given that $A_2^S \sim A_2^S$. By inversion on $\text{DirCons}$ there exist $A$ and $B$ such that $A \sqsubseteq A_1^S$ and $A \leq B$ and $B \sqsubseteq A_2^S$. By Lemma 35 (Precision for static types), $A = A_1^S$ and $B = A_2^S$. Therefore, $A \leq B$ is equivalent to $A_1^S \leq A_2^S$.

Theorem 5 (Static soundness and completeness).

1. Soundness:
   (a) If $\Gamma^S \vdash e^S \Leftarrow A^S_0$ then $\Gamma^S \vdash e^S \Leftarrow A^S_0$.
   (b) If $\Gamma^S \vdash e^S \Rightarrow A^S_0$ then $\Gamma^S \vdash e^S \Rightarrow A^S_0$.

2. Completeness:
   (a) If $\Gamma^S \vdash e^S \Leftarrow A^S_0$ then $\Gamma^S \vdash e^S \Leftarrow A^S_0$.
   (b) If $\Gamma^S \vdash e^S \Rightarrow A^S_0$ then $\Gamma^S \vdash e^S \Rightarrow A^S_0$.

Proof.

1. By induction on the structure of the given derivation.

   Case $\text{StVar}$ Apply rule $\text{SynVar}$

   Case $\Gamma^S \vdash e^S \Rightarrow A^S_0$ $A^S_0 \leq_s A^S$

   $\Gamma^S \vdash e^S \Leftarrow A^S_0$ Subderivation
   $A^S_0 \leq_s A^S$ Subderivation
   $\Gamma^S \vdash e^S \Rightarrow A^S_0$ By the induction hypothesis
   $A^S_0 \leq A^S$ By Lemma 37 (Equivalence for static subtyping)
   $A^S_0 \sim A^S$ By Lemma 38 (Subtyping obeys directed consistency)
   $\Gamma^S \vdash e^S \Leftarrow A^S$ By rule $\text{ChkSub}$

   Case $\text{StAnno}$ Use the induction hypothesis and apply rule $\text{SynAnno}$

   Case $\text{StUnitIntro}$ Apply rule $\text{ChkUnitIntro}$

   Case $\Gamma^S \vdash e_i^S \Leftarrow A^S_1$ $\delta_i^S \leq S 

   $\Gamma^S \vdash e_i^S \Leftarrow A^S_1$ Subderivation
   $\delta_i^S \leq S$ Subderivation
   $\Gamma^S \vdash e_i^S \Leftarrow A^S_1$ By the induction hypothesis
   $\delta_i^S \leq S$ By definition of $\leq$

   Case $\text{StSumIntro}$ Use the induction hypothesis, the definition of $\Rightarrow$ and apply rule $\text{ChkSumIntro}$

   Case $\text{StSumElim1}$ Use the induction hypothesis, the definition of $\Rightarrow$ and apply rule $\text{ChkSumElim1}$

   Case $\text{StSumElim2}$ Use the induction hypothesis, the definition of $\Rightarrow$ and apply rule $\text{ChkSumElim2}$

   Case $\chi \rightarrow \text{Intro}$ Use the induction hypothesis and apply rule $\text{ChkIntro}$

   Case $\text{StIntro}$ Use the induction hypothesis and apply rule $\text{SynIntro}$

   Case $\text{StElim}$ Use the induction hypothesis and apply rule $\text{SynElim}$
2. By induction on the structure of the given derivation.

- **Case SynVar**: Apply rule \texttt{StVar}.

- **Case**
  
  \[
  \Gamma^S \vdash e^S : \tilde{A}_2^S \quad \quad \text{ChkSub}
  \]
  
  \[
  \Gamma^S \vdash e^S : \tilde{A}_0^S \quad \quad \text{Subderivation}
  \]
  
  \[
  \tilde{A}_0^S \sim \tilde{A}_2^S \quad \quad \text{Subderivation}
  \]
  
  \[
  \tilde{A}_2^S \leq \tilde{A}_2^S \quad \quad \text{By Lemma~38 (Directed consistency for static types)}
  \]
  
  \[
  \Gamma^S \vdash e^S : \tilde{A}_0^S \quad \quad \text{By the induction hypothesis}
  \]
  
  \[
  \Gamma^S \vdash e^S : \tilde{A}_2^S \quad \quad \text{By rule StSub}
  \]

- **Case SynAnno**: Use the induction hypothesis and apply rule \texttt{StAnno}.

- **Case** \texttt{ChkUnitIntro}: Apply rule \texttt{StUnitIntro}.

- **Case**
  
  \[
  \Gamma^S \vdash e_i^S \leftarrow A_i^S \quad \quad \text{ChkSumIntro}
  \]
  
  \[
  \Gamma^S \vdash e_i^S : A_i^S \quad \quad \text{Subderivation}
  \]
  
  \[
  +_i^S \leq \delta^S \quad \quad \text{Subderivation}
  \]
  
  \[
  \Gamma^S \vdash e_i^S : A_i^S \quad \quad \text{By the induction hypothesis}
  \]
  
  \[
  +_i^S \leq \delta^S \quad \quad \text{By Lemma~32 (Static looseness)}
  \]
  
  \[
  \Gamma^S \vdash \text{inj}_i^S, e_i^S : (A_i^S \delta^S A_2^S) \quad \quad \text{By rule StSumIntro}
  \]

- **Case**
  
  \[
  \delta^S \Rightarrow +_i^S \quad \quad \text{Subderivation}
  \]
  
  \[
  \Gamma^S \vdash \text{case}(e_i^S, \text{inj}_i^S) : A_i^S \quad \quad \text{ChkSumElim1}
  \]
  
  \[
  \Gamma^S \vdash e_i^S : (A_i^S \delta^S A_2^S) \quad \quad \text{Subderivation}
  \]
  
  \[
  +_i^S \leq \delta^S \quad \quad \text{Subderivation}
  \]
  
  \[
  \Gamma^S \vdash e_i^S : (A_i^S \delta^S A_2^S) \quad \quad \text{By the induction hypothesis}
  \]
  
  \[
  \delta^S \Rightarrow +_i^S \quad \quad \text{By the induction hypothesis}
  \]
  
  \[
  +_i^S \leq \delta^S \quad \quad \text{By Lemma~33 (Static looseness, II)}
  \]
  
  \[
  \Gamma^S \vdash \text{case}(e_i^S, \text{inj}_i^S) : A_i^S \quad \quad \text{By rule StSumElim1}
  \]

- **Case** \texttt{ChkSumElim2}: Use the induction hypothesis, the definition of \(\leq_S\) and apply rule \texttt{StSumElim2}.

- **Case** \texttt{Chk—Intro}: Use the induction hypothesis and apply rule \texttt{St—Intro}.

- **Case** \texttt{Syn—Elim}: Use the induction hypothesis and apply rule \texttt{St—Elim}.

\[\square\]

### D.4 Properties of the Dynamic System

**Lemma 39** (Subtyping for dynamic types).

If \(A_1^D \leq A_2^D\) then \(A_1^D = A_2^D\).

**Proof.** By induction on the structure of \(A_1^D\).

- **Case** \(A_1^D = \text{Unit}\): By the definition of subtyping, \(A_1^D = \text{Unit}\) only.

- **Case** \(A_1^D = A_1^D +^S A_2^D\):
  
  \[
  A_1^D +^S A_2^D \leq A_2^D \quad \quad \text{Given}
  \]
  
  \[
  A_2^D = A_1^D +^S A_2^D \quad \quad \text{From the definition of } \leq
  \]
  
  \[
  A_2^D \leq A_1^D \quad \quad \text{"}
  \]
  
  \[
  A_2^D \leq A_1^D \quad \quad \text{"}
  \]
  
  \[
  A_2^D = A_1^D \quad \quad \text{By the induction hypothesis}
  \]
  
  \[
  A_2^D = A_1^D \quad \quad \text{By the induction hypothesis}
  \]
  
  \[
  A_2^D = A_1^D \quad \quad \text{By definition of } =
  \]

- **Case** \(A_1^D = A_1^D \Rightarrow A_2^D\): Similar to the previous case. 

\[\square\]

**Lemma 40** (Precision for dynamic types).

If \(A_1 \subseteq A_2^D\) then \(A_1 = A_2^D\).
Proof. By induction on the structure of $A_D^0$.

- **Case** $A_D^0 = \text{Unit}$. By the definition of imprecision, $A_1 = \text{Unit}$ only.
- **Case** $A_D^0 = A_D^{01} + \ast A_D^{02}$:
  
  $A_1 \subseteq A_D^0$ Given
  
  $A_1 = A_{11} + \ast A_{21}$ From the definition of $\subseteq$
  
  $A_{11} \subseteq A_D^{01}$
  
  $A_{21} \subseteq A_D^{02}$
  
  $A_{11} = A_{12}$ By the induction hypothesis
  
  $A_{21} = A_{22}$ By the induction hypothesis
  
  $A_1 = A_1^2$ By definition of $= \\

- **Case** $A_D^0 = A_D^{11} \rightarrow A_D^{12}$: Similar to the previous case. \[\square\]

**Lemma 41** (Directed consistency for dynamic types). If $A_D^0 \sim A_D^0$ then $A_D^0 = A_D^0$. 

*Proof.* It is given that $A_D^0 \sim A_D^0$. By inversion on $\text{DirCons}$ there exist $A$ and $B$ such that $A \subseteq A_D^0$ and $A \subseteq B$ and $B \subseteq A_D^0$. By Lemma 40 (Precision for dynamic types), $A = A_D^0$ and $B = A_D^0$. Therefore, $A \subseteq B$ is equivalent to $A_D^0 \subseteq A_D^0$. By Lemma 39 (Subtyping for dynamic types), $A_D^0 = A_D^0$. \[\square\]

**Theorem 15** (Dynamic soundness and completeness).

1. (a) If $\Gamma^D \vdash e^D \sqsubseteq A^D$ then $\Gamma^D \vdash e^D \sqsubseteq A^D$.
2. (b) If $\Gamma^D \vdash e^D \Rightarrow A^D$ then $\Gamma^D \vdash e^D \Rightarrow A^D$.

*Proof.*

1. By induction on the structure of the given $\vdash^D$-derivation.

   - **Case** $\text{DVar}$ Apply rule $\text{SynVar}$
   - **Case** $\text{DSub}$ Use the induction hypothesis, reflexivity of directed consistency, and apply rule $\text{ChkCSub}$
   - **Case** $\text{DUnitIntro}$ Apply rule $\text{ChkUnitIntro}$
   - **Case** $\text{DVar}^D \vdash e^D \sqsubseteq A^D$ Subderivation
     
     $\Gamma^D \vdash e^D \sqsubseteq A^D$ By the induction hypothesis

     $\Gamma^D \vdash e^D \sqsubseteq A^D$ By definition of $\sqsubseteq$

     $\Gamma^D \vdash e^D \sqsubseteq (A^D + \ast A_D^0)$ By rule $\text{ChkSumIntro}$

   - **Case** $\text{DVar}^D \vdash e^D \sqsubseteq A^D$ Use the induction hypothesis, the definition of $\Rightarrow$ and apply rule $\text{ChkSumElim1}$
   - **Case** $\text{DVar}^D \vdash e^D \sqsubseteq A^D$ Use the induction hypothesis, the definition of $\Rightarrow$ and apply rule $\text{ChkSumElim2}$
   - **Case** $\text{DVar}^D \vdash e^D \sqsubseteq A^D$ Use the induction hypothesis and apply rule $\text{ChkIntro}$
   - **Case** $\text{DVar}^D \vdash e^D \sqsubseteq A^D$ Use the induction hypothesis and apply rule $\text{SynElim}$

2. By induction on the structure of the given $\vdash^D$-derivation.

   - **Case** $\text{DVar}^D \vdash e^D \Rightarrow A^D$ Simple derivation
     
     $\Gamma^D \vdash e^D \Rightarrow A^D$ $A_0 \sim A_D^0$ Subderivation

     $\Gamma^D \vdash e^D \Rightarrow A^D$ By Lemma 41 (Directed consistency for dynamic types)

     $\Gamma^D \vdash e^D \Rightarrow A^D$ Subderivation

     $\Gamma^D \vdash e^D \Rightarrow A^D$ By the induction hypothesis

     $\Gamma^D \vdash e^D \Rightarrow A^D$ Equivalent

     $\Gamma^D \vdash e^D \Rightarrow A^D$ By rule $\text{DSub}$

   - **Case** $\text{SynAnno}$ Use the induction hypothesis and apply rule $\text{DAAnno}$
   - **Case** $\text{DVar}^D \vdash e^D \Rightarrow A^D$ Simple derivation
     
     $\Gamma^D \vdash e^D \Rightarrow A^D$ $A_0 \sim A_D^0$ Subderivation

     $\Gamma^D \vdash e^D \Rightarrow A^D$ By Lemma 41 (Directed consistency for dynamic types)

     $\Gamma^D \vdash e^D \Rightarrow A^D$ Subderivation

     $\Gamma^D \vdash e^D \Rightarrow A^D$ By the induction hypothesis

     $\Gamma^D \vdash e^D \Rightarrow A^D$ Equivalent

     $\Gamma^D \vdash e^D \Rightarrow A^D$ By rule $\text{DSub}$

   - **Case** $\text{ChkUnitIntro}$ Apply rule $\text{DUnitIntro}$
By induction on the structure of $T$. 

**Proof.**
1. By case analysis on $T' \leq \text{Unit}$.
   - **Case** $\text{Unit} \leq \text{Unit}$: Immediate that $T' = \text{Unit}$.
2. Symmetric to part 1.
3. By case analysis on $T' \leq T_1 \phi T_2$.
   - **Case** $T_1' \phi' T_2' \leq T_1 \phi T_2$: Immediate as $T' = T_1' \phi' T_2'$ and subderivations are $T_1' \leq T_1$ and $T_2' \leq T_2$ and $\phi' \leq \phi$.
4. Symmetric to part 3.
5. By case analysis on $T' \leq T_1 \rightarrow T_2$.
   - **Case** $T_1' \rightarrow T_2' \leq T_1 \rightarrow T_2$: Immediate as $T' = T_1' \rightarrow T_2'$ and subderivations are $T_1 \leq T_1'$ and $T_2' \leq T_2$.
6. Symmetric to part 5. 

**Lemma 43** (Reflexivity of subtyping).

For all types $T$, it is the case that $T \leq T$.

**Proof.** By induction on the structure of $T$.

- **Case** $T = \text{Unit}$: By the definition of subtyping, $T \leq T$.
- **Case** $T = T_1 \phi T_2$: By the induction hypothesis, $T_1 \leq T_1$ and $T_2 \leq T_2$. By the reflexivity of subsum, $\phi \leq \phi$. Thus, by the definition of subtyping, $T \leq T$.
- **Case** $T = T_1 \rightarrow T_2$: By the induction hypothesis, $T_1 \leq T_1$ and $T_2 \leq T_2$. Thus, by the definition of subtyping, $T \leq T$. 

**Lemma 44** (Transitivity of subtyping).

If $T_1 \leq T_2$ and $T_2 \leq T_3$ then $T_1 \leq T_3$.

**Proof.** By induction on the structure of $T_2$.

- **Case** $T_2 = \text{Unit}$.
  - $T_1 \leq \text{Unit}$ : Given
  - $T_1 = \text{Unit}$ : By Lemma \ref{lemma:subtyping-inversion}(Subtyping inversion)
  - $T_3 = \text{Unit}$ : By Lemma \ref{lemma:reflexivity-of-subtyping}(Reflexivity of subtyping)
  - $T_1 \leq T_3$ : Equivalent
- **Case** $T_2 = T_{12} \phi_2 T_{22}$.
  - $T_1 \leq T_{12} \phi_2 T_{22}$ : Given
  - $T_1 = T_{11} \phi_1 T_{12}$ : By Lemma \ref{lemma:subtyping-inversion}(Subtyping inversion)
  - $T_{11} \leq T_{12}$ : "
  - $T_{21} \leq T_{22}$ : "
  - $\phi_1 \leq \phi_2$ : "
  - $T_{12} \phi_2 T_{22} \leq T_3$ : Given
  - $T_3 = T_{13} \phi_3 T_{23}$ : By Lemma \ref{lemma:subtyping-inversion}(Subtyping inversion)
  - $T_{12} \leq T_{13}$ : "
  - $T_{22} \leq T_{23}$ : "
  - $\phi_2 \leq \phi_3$ : "
Proof.

Corollary 45 (Subtyping inversion).
1. If \( T' \phi' T'' \leq T \phi T \) then \( T'_1 \leq T_1 \) and \( T'_2 \leq T_2 \) and \( \phi' \leq \phi \).
2. If \( T' \rightarrow T'_2 \leq T_1 \rightarrow T_2 \) then \( T_1 \leq T'_1 \) and \( T'_2 \leq T_2 \).

Proof:
1. Let \( T' = T'_i \phi' T'' \). We are given \( T' \leq T_1 \phi T_2 \). Therefore, by Lemma 42 (Subtyping inversion), \( T'_i \leq T_1 \) and \( T'_2 \leq T_2 \) and \( \phi' \leq \phi \).
2. Let \( T' = T_i \rightarrow T'_2 \). We are given \( T' \leq T_1 \rightarrow T_2 \). Therefore, by Lemma 42 (Subtyping inversion), \( T_i \leq T'_1 \) and \( T'_2 \leq T_2 \).

D.5.2 Values

Lemma 46 (Value inversion).
1. If \( i \vdash W : T \) and \( T \leq (T_1 + T_2) \) then \( W = \text{inj}_i W_i \) and \( i : W_i : T_1 \).
   Moreover, if \( T \leq (T_1 +_k T_2) \) then \( i = k \).
2. If \( i \vdash W : T \) and \( T \leq (T_1 \rightarrow T_2) \) then \( W = \lambda x. M \) and \( \lambda x. x : T_1 \vdash M : T_2 \).

Proof:
1. By induction on the structure of the derivation of \( i \vdash W : T \).

   • Case \[ \text{TVar} \] Impossible because context \( \Theta = i \) is empty.

   • Case \[ \vdash W : T' \]
     \[ \vdash W : T \]
     \[ \text{TSub} \]
     \[ T' \leq T \quad \text{Subderivation} \]
     \[ T \leq T_1 + T_2 \quad \text{Given} \]
     \[ T' \leq T_1 + T_2 \quad \text{By Lemma 44 (Transitivity of subtyping).} \]
     Immediate from the induction hypothesis.

   • Cases \[ \text{TCast, TMatchfail} \] Impossible because the subject term is not a value.

   • Case \[ \text{TUnitIntro} \] Impossible because \( T = \text{Unit} \) cannot be a subtype of \( T_1 + T_2 \).

   • Case \[ \vdash W_i : T'_i \]
     \[ \vdash \text{inj}_i W_i : (T'_i + T_2) \]
     \[ \text{T+iIntro} \]
     By the definition of values \( W_i \), we know that \( W_i \) is a value and \( W = \text{inj}_i W_i \).
     \[ T'_i + T_2 \leq T_1 + T_2 \quad \text{Given} \]
     \[ T'_i \leq T_1 \quad \text{By Corollary 45} \]
     \[ \vdash W_i : T_1 \quad \text{Subderivation} \]
     \[ i \leq k \quad \text{By rule TSub} \]
     \[ T'_i + T_2 \leq T_1 + k T_2 \quad \text{Suppose} \]
     \[ k \leq k \quad \text{By Corollary 45} \]
     \[ i = k \quad \text{From definition of \( \leq \) } \]
• Cases $T\rightarrow\text{Elim}$: Impossible because the subject term is not a value.
• Case $T\rightarrow\text{Intr}$: Impossible because $T = T'_1 \rightarrow T'_2$ cannot be a subtype of $T_1 + T_2$.
• Case $T\rightarrow\text{UnitIntro}$: Impossible because the subject term is not a value.

2. By induction on the structure of the derivation of $\vdash W : T$.

• Case $W\rightarrow\text{Var}$: Impossible because context $\Theta = \cdot$ is empty.

• Case $\vdash W : T' \quad T' \leq T$
  $\vdash W : T$  Subderivation

2. By induction on the structure of the derivation of $\vdash W : T$.

• Case $W\rightarrow\text{Var}$: Impossible because context $\Theta = \cdot$ is empty.

• Case $\vdash W : T' \quad T' \leq T$
  $\vdash W : T$  Subderivation

Immediate from the induction hypothesis.

• Cases $W\rightarrow\text{Cast}, W\rightarrow\text{Matchfail}$: Impossible because the subject term is not a value.
• Case $W\rightarrow\text{UnitIntro}$: Impossible because $T = \text{Unit}$ cannot be a subtype of $T_1 + T_2$.
• Case $W\rightarrow\text{Intr}$: Impossible because $T = T'_1 \rightarrow T'_2$ cannot be a subtype of $T_1 + T_2$.
• Case $W\rightarrow\text{UnitIntro}$: Impossible because the subject term is not a value.

By definition of values $W$, we know that $W = \lambda x. M$.

$T'_1 \rightarrow T'_2 \leq T_1 \rightarrow T_2$  Given
$T_1 \leq T'_1$  By Corollary 45
$T'_2 \leq T_2$  "
$\vdash x : T'_1 \vdash M : T'_2$  Subderivation
$\vdash x : T_1 \vdash M : T_2$  By Lemma 50 (Context Strengthening)
$\vdash \lambda x. M : (T_1 \rightarrow T_2)$.  By rule $T\rightarrow\text{Intro}$

$\vdash W : (T_1 + T_2)$ then $W = \text{inj}_1 W_i$ and $\vdash W_i : T_i$.

Proof. Let $T = T_1 + T_2$.

$T_1 \leq T_1$  By Lemma 43 (Reflexivity of subtyping)
$T_2 \leq T_2$  By Lemma 43 (Reflexivity of subtyping)
$+ \leq +$  By definition of $\leq$
$T \leq T_1 + T_2$  By definition of $\leq$
$\vdash W : T$  Given
$W = \text{inj}_i W_i$  By Lemma 46 (Value inversion)
$\vdash W_i : T_i$  "
$(T \leq T_1 + T_2)$ implies $(k = i)$  "
$T \leq T$  By Lemma 43 (Reflexivity of subtyping)
$i = k$  Implication
$W = \text{inj}_1 W_i$  Equivalent
$\vdash W_i : T_i$  Equivalent

$\vdash W : (T_1 + T_2)$ then $W = \lambda x. M_0$ and $\vdash W_i : T_i$.

Proof. By Lemma 46 (Value inversion) with $T = T_1 + T_2$, using Lemma 43 (Reflexivity of subtyping).

$\vdash W : (T_1 \rightarrow T_2)$ then $W = \lambda x. M_0$ and $\vdash x : T_1 \vdash M_0 : T_2$.

Proof. By Lemma 46 (Value inversion) with $T = T_1 \rightarrow T_2$, using Lemma 43 (Reflexivity of subtyping).
D.5.3 Typing and Evaluation Contexts

Lemma 50 (Context Strengthening).
If \( \Theta, y : T' \vdash M : T_0 \) and \( T \leq T' \) then \( \Theta, y : T \vdash M : T_0 \).

Proof. By induction on the structure of the derivation of \( \Theta, y : T' \vdash M : T_0 \).

- Case \( (\Theta, y : T')(M) = T_0 \) \( \Theta, y : T' \vdash M : T_0 \) \( \text{TVar} \)
  Either \( M = y \), or \( M \neq y \).
  In the first case:
  \( (\Theta, y : T)(M) = T_0 \) Premise
  \( T' = T_0 \) By definition
  \( \Theta, y : T \vdash y : T \) By rule \( \text{TVar} \)
  \( T \leq T' \) Given
  \( \Theta, y : T \vdash y : T' \) By rule \( \text{TSub} \)
  \( \Theta, y : T \vdash M : T_0 \) By above equalities

In the second case:
\( \Theta, y : T \vdash M : T_0 \) By rule \( \text{TVar} \)

- Case \( \text{TSub} \) Use the induction hypothesis and apply rule \( \text{TSub} \)
- Case \( \text{TCast} \) Use the induction hypothesis and apply rule \( \text{TCast} \)
- Case \( \text{TMatchfail} \) Immediate from rule \( \text{TMatchfail} \)
- Case \( \text{TUnitIntro} \) Immediate from rule \( \text{TUnitIntro} \)
- Case \( \text{T+Intro} \) Use the induction hypothesis and apply rule \( \text{T+Intro} \)
- Case \( \text{T+Elim} \) Use the induction hypothesis and apply rule \( \text{T+Elim} \)
- Case \( \text{T=Intro} \) Use the induction hypothesis and apply rule \( \text{T=Intro} \)
- Case \( \text{T=Elim} \) Use the induction hypothesis and apply rule \( \text{T=Elim} \) \( \square \)

Lemma 51 (Substitution).
If \( \Theta, x : T' \vdash M : T \) and \( \cdot \vdash W : T' \) then \( \Theta \vdash [W/x]M : T \).

Proof. By induction on the structure of the derivation of \( \Theta, x : T' \vdash M : T \).

- Case \( \text{TVar} \) Use the definition of substitution, well-formedness of \( \Theta \), and rule \( \text{TVar} \)
- Case \( \text{TSub} \) Use the definition of substitution, the induction hypothesis and apply rule \( \text{TSub} \)
- Case \( \text{TCast} \) Use the definition of substitution, the induction hypothesis and apply rule \( \text{TCast} \)
- Case \( \text{TMatchfail} \) Use the definition of substitution and apply rule \( \text{TMatchfail} \)
- Case \( \text{TUnitIntro} \) Use the definition of substitution and apply rule \( \text{TUnitIntro} \)
- Case \( \text{T+Intro} \) Use the definition of substitution, the induction hypothesis and apply rule \( \text{T+Intro} \)
- Case \( \text{T+Elim} \) Use the definition of substitution, the induction hypothesis and apply rule \( \text{T+Elim} \)
- Case \( \text{T=Intro} \) Use the definition of substitution, the induction hypothesis and apply rule \( \text{T=Intro} \)
- Case \( \text{T=Elim} \) Use the definition of substitution, the induction hypothesis and apply rule \( \text{T=Elim} \) \( \square \)

Lemma 52 (Evaluation context typing).
If \( \Theta \vdash E[M_0] : T \) then there exists \( T_0 \) such that \( \Theta \vdash M_0 : T_0 \).

Proof. By induction on the structure of the derivation of \( \Theta \vdash E[M_0] : T \).

- Case \( \text{TVar} \) Immediate as \( E[M_0] = M_0 \), so \( T_0 = T \).
- Case \( \text{TSub} \) Immediate from the induction hypothesis.
- Case \( \text{TCast} \) Immediate from the induction hypothesis.
- Case \( \text{TMatchfail} \) Immediate as \( E[M_0] = M_0 \), so \( T_0 = T \).
- Case \( \text{TUnitIntro} \) Immediate as \( E[M_0] = M_0 \), so \( T_0 = T \).
- Case \( \text{T+Intro} \) Immediate from the induction hypothesis.
- Case \( \text{T+Elim} \) Immediate from the induction hypothesis.
- Case \( \text{T=Intro} \) Immediate from the induction hypothesis.
- Case \( \text{T=Elim} \) Immediate as \( E[M_0] = M_0 \), so \( T_0 = T \).
Lemma 53 (Evaluation context replacement).
If $\Theta \vdash \varepsilon[M_0] : T$ and $\Theta \vdash M_0 : T_0$ and $\Theta \vdash M'_0 : T_0$ then $\Theta \vdash \varepsilon[M'_0] : T$.

Proof. By induction on the structure of the derivation of $\Theta \vdash \varepsilon[M_0] : T$.

- Case $\mathsf{TVar}$: Immediate as $\varepsilon[M_0] = M_0$, so $T_0 = T$ and $\varepsilon[M'_0] = M'_0$.
- Case $\mathsf{TSub}$: Use the induction hypothesis and apply rule $\mathsf{TSub}$.
- Case $\mathsf{TCast}$: Use the induction hypothesis and apply rule $\mathsf{TCast}$.
- Case $\mathsf{TMatchfail}$: Immediate as $\varepsilon[M_0] = M_0$, so $T_0 = T$ and $\varepsilon[M'_0] = M'_0$.
- Case $\mathsf{TUnitIntro}$: Immediate as $\varepsilon[M_0] = M_0$, so $T_0 = T$ and $\varepsilon[M'_0] = M'_0$.
- Case $\mathsf{T+Intro}$: Use the induction hypothesis and apply rule $\mathsf{T+Intro}$.
- Case $\mathsf{T+Elim}$: Use the induction hypothesis and apply rule $\mathsf{T+Elim}$.
- Case $\mathsf{T=Intro}$: Immediate as $\varepsilon[M_0] = M_0$, so $T_0 = T$ and $\varepsilon[M'_0] = M'_0$.

D.5.4 Type Safety

Lemma 54 (Type preservation under reduction).
If $\vdash M : T$ and $M \rightsquigarrow_R M'$ then $\vdash M' : T$.

Proof. By induction on the structure of the derivation of $\vdash M : T$.

- Case $\mathsf{TVar}$: Impossible because the context $\Theta = \cdot$ is empty.
- Case $\mathsf{TSub}$: Use the induction hypothesis and apply rule $\mathsf{TSub}$.
- Case $\cdot \vdash M_0 : (T_1 \phi' T_2)$: Proceed by case analysis on $M \rightsquigarrow_R M'$.

- Case $\phi' \leq \phi$: Impossible because $\phi'$ is empty.

- Case $\mathsf{T+Intro}$: Use the induction hypothesis and apply rule $\mathsf{T+Intro}$.
- Case $\mathsf{T+Elim}$: Impossible because $\mathsf{matchfail} \not\vdash_R M'$ for any $M'$.
- Case $\mathsf{TUnitIntro}$: Impossible because $\emptyset \not\vdash_R M'$ for any $M'$.
- Case $\mathsf{T=Intro}$: Impossible because $M = \mathsf{inj}_i M_0 \not\vdash_R M'$ for any $M'$.
• Case \( \vdash M_0 : T_1 + T_2 \quad \therefore, x : T_i \vdash M_i : T \) \[ T+\text{Elim} \]

Proced by case analysis on \( M \mapsto_R M' \).

• Case \[ \text{ReduceCase1} \]

• Case \[ \text{ReduceCase2} \]

• Case \[ \text{T+Elim} \] Similar to the \( \text{T+Elim} \) case. Apply Corollary 48 instead of Corollary 47 when considering the \( \text{ReduceCase2} \) case.

• Case \[ \text{T=Intro} \] Impossible because \( M = \lambda x. M_0 \not\mapsto_R M' \) for any \( M' \).

• Case \[ \text{T=Elim} \]

Proced by case analysis on \( M \mapsto_R M' \).

\[ \text{Theorem 6 (Type preservation).} \]

If \( \vdash M : T \) and \( M \mapsto M' \) then \( \vdash M' : T \).

\[ \text{Proof.} \] By case analysis on \( M \mapsto M' \).

\[ \text{Case} \]

\[ \text{StepContext} \]

\[ \text{StepMatchfail} \]

Immediately by \[ \text{TMatchfail} \]

\[ \text{Theorem 7 (Progress).} \]

If \( \vdash M : T \) then either (a) \( M \) is a value, or (b) there exists \( M' \) such \( M \mapsto M' \), or (c) \( M = \text{matchfail} \).

\[ \text{Proof.} \] By induction on the structure of the derivation of \( \vdash M : T \).

• Case \[ \text{TVar} \] Impossible, because the context \( \Theta \) is empty.

• Case \[ \text{TSub} \] Immediate by the induction hypothesis.

• Case \[ \text{TCast} \]

We have \( M = (\phi \equiv \phi')M_0 \) and \( T = T_1 \phi T_2 \) where \( \vdash M_0 : (T_1 \phi' T_2) \).

By the induction hypothesis, either \( M_0 \) is a value or there exists \( M'_0 \) such that \( M_0 \mapsto M'_0 \).

In the first case, we need to consider all possible assignments to \( \phi' \) and \( \phi \).

Suppose \( \phi' \leq \phi \), then \( M \mapsto M_0 \).

Suppose \( \phi' = +_i \) and \( \phi = +_k \) where \( i \neq k \), then \( M_0 = \text{inj}_i W \) by Corollary 47 so \( M \mapsto \text{matchfail} \).
Suppose \( \phi' = + \) and \( \phi = +_i \), then \( M_0 = \text{inj}_k W \) by Corollary [48]. Proceed by cases analysis on \( i \), if \( i = k \) then \( M \rightarrow M_0 \), otherwise \( M \rightarrow \text{matchfail} \).

In the second case, \( \langle \phi \neq \phi' \rangle M_0 \rightarrow \langle \phi \neq \phi' \rangle M'_0 \).

**Case TUnitIntro** We have \( M = () \), a value, which is alternative (a).

**Case TMatchfail** We have \( M = \text{matchfail} \), which is alternative (c).

**Case T+;Intro**

\[
\begin{align*}
\text{We have } M &= \text{inj}_1 M_0 \\
T &= T_1 + T_2 & \text{where } \cdot \vdash M_0 : T_i.
\end{align*}
\]

By the induction hypothesis, either \( M_0 \) is a value or there exists \( M'_0 \) such that \( M_0 \rightarrow M'_0 \).

In the first case, \( \text{inj}_1 M_0 = M \) is a value.

In the second case, \( \text{inj}_1 M_0 \rightarrow \text{inj}_1 M'_0 \).

**Case T+;Elim**

\[
\begin{align*}
\text{We have } M &= \text{case}(M_0, \text{inj}_1 x.M_1) & \text{where } \cdot \vdash M_0 : T_1 + T_2 & \text{and } \cdot : T_1 \vdash M_1 : T.
\end{align*}
\]

By the induction hypothesis, either \( M_0 \) is a value or there exists \( M'_0 \) such that \( M_0 \rightarrow M'_0 \).

In the first case, \( M_0 = \text{inj}_1 W \) by Corollary [47] so \( \text{case}(M_0, \text{inj}_1 x.M_1) \rightarrow [W/x]M_1 \).

In the second case, \( \text{case}(M_0, \text{inj}_1 x.M_1) \rightarrow \text{case}(M'_0, \text{inj}_1 x.M_1) \).

**Case T+;Elim** Similar to the T+;Intro case, using Corollary [48] instead of Corollary [47]

**Case T;Intro** We have \( M = \lambda x. M_0 \), a value.

**Case T;Elim**

\[
\begin{align*}
\text{We have } M &= M_1 M_2 & \text{where } \cdot \vdash M_1 : T_1 & \rightarrow T_2 & \text{and } \cdot \vdash M_2 : T_1.
\end{align*}
\]

By the induction hypothesis, either \( M_1 \) is a value or there exists \( M'_1 \) such that \( M_1 \rightarrow M'_1 \).

In the first case, \( M_1 = \lambda x. M_0 \) by Corollary [49]

By the induction hypothesis, either \( M_2 \) is a value or there exists \( M'_2 \) such that \( M_2 \rightarrow M'_2 \).

In the first subcase, \( M_2 \) is a value, so \( \lambda x. M_2 ) M_2 \rightarrow [M_2/x]M_0 \).

In the second subcase, \( \lambda x. M_0 \) \( M_2 \rightarrow [\lambda x. M_0] M'_2 \).

In the second case, \( M_1 M_2 \rightarrow M'_1 M'_2 \).

**Theorem 8** (matchfail-freeness).

If \( M \) is cast-free and matchfail-free and \( M \rightarrow M' \) then \( M' \) is cast-free and matchfail-free.

**Proof.**

By induction on the derivation of \( M \rightarrow M' \).

By the assumption that \( M \) is matchfail-free, rule \text{StepMatchfail} is impossible. Therefore, the derivation is by \text{StepContext} with subderivation \( M_0 \rightarrow \Rightarrow M'_0 \) where \( M = \epsilon[M_0] \) and \( M' = \epsilon[M'_0] \).

- **Cases ReduceUpcast** \text{ReduceCastSuccess} \text{ReduceCastFailure} In these cases, \( M_0 \) contains a cast, contradicting the assumption that \( M = \epsilon[M_0] \) is cast-free. Hence, these cases are impossible.

- **Case ReduceCase1**

\[
\begin{align*}
\text{We have } M_0 &= \text{case}\langle \text{inj}_1 W, \text{inj}_1 x.M_1 \rangle & \text{and } M'_1 &= [W/x]M_1.
\end{align*}
\]

Since \( M_0 \) is cast- and matchfail-free, its subterms \( W \) and \( M_1 \) are cast- and matchfail-free.

Therefore, \( [W/x]M_1 \) is cast- and matchfail-free.

- **Cases ReduceCase2** \text{ReduceCase1} Similar to the ReduceCase1 case.

D.5.5 Precision

**Lemma 55** (Precision on values).

If \( W' \equiv M \) then \( M = W \) for some value \( M \).

**Proof.**

By induction on the structure of the derivation of \( W' \equiv M \).

- **Case () \equiv M:** From definition of \( \equiv \), it is immediate that \( M = () \), a value.

- **Case x \equiv M:** From definition of \( \equiv \), it is immediate that \( M = x \), a value.

- **Case \lambda x. M_0 \equiv M:** From definition of \( \equiv \), it is immediate that \( M = \lambda x. M_0 \), a value.

- **Case inj_1 W_0 \equiv M:** From definition of \( \equiv \), \( M = \text{inj}_1 M_0 \) and \( W_0 \equiv M_0 \). By the induction hypothesis, \( M_0 = W_0 \) for some value \( W_0 \).

Therefore, \( M = \text{inj}_1 W_0 \), a value.

**Lemma 56** (Substitution preserves precision).

If \( M' \equiv M \) and \( W' \equiv W \) then \( [W'/x]M' \equiv [W/x]M \).

**Proof.**

By induction on the structure of the derivation of \( M' \equiv M \). All cases are immediate by the induction hypothesis, the definition of substitution, and the definition of \( \equiv \).

**Lemma 57** (Precision inversion on evaluation contexts).

If \( \epsilon' [M'_0] \equiv M \) then there exists \( \epsilon \) and \( M_0 \) such that \( M = \epsilon[M_0] \) and \( M'_0 \equiv M_0 \).

**Proof.**

Proceed by induction on the structure of \( \epsilon' \).

- **Case \epsilon' = \text{[]}** Choose \( \epsilon = \text{[]} \) and \( M_0 = M \) then \( M'_0 \equiv M_0 \) is given.
• **Case $\mathcal{E}' = \text{inj}_i \mathcal{E}'_0$:**

  \[
  \mathcal{E}'[\mathcal{M}'_0] \preceq M \\
  \text{inj}_i \mathcal{E}'_0[\mathcal{M}'_0] \preceq M \\
  \mathcal{M} = \text{inj}_i \mathcal{M}_1 \\
  \mathcal{E}'[\mathcal{M}'_0] \preceq \mathcal{M}_1 \\
  \mathcal{M}_1 = \mathcal{E}_0[\mathcal{M}_0] \\
  \mathcal{M}'_0 \preceq \mathcal{M}_0 \\
  \mathcal{M} = \text{inj}_i \mathcal{E}_0[\mathcal{M}_0] \\
  \] 

  From the definition of $\preceq$

  By the induction hypothesis

  By the induction hypothesis

• **Case $\mathcal{E}' = \mathcal{E}_0 \mathcal{M}_2$:**

  \[
  \mathcal{E}'[\mathcal{M}_0] \preceq M \\
  \mathcal{E}_0[\mathcal{M}_0] \mathcal{M}_2 \preceq M \\
  \mathcal{M} = \mathcal{M}_1 \mathcal{M}_2 \\
  \mathcal{E}_0[\mathcal{M}_0] \mathcal{M}_1 \preceq \mathcal{M}_1 \\
  \mathcal{M}_1 = \mathcal{E}_0[\mathcal{M}_0] \\
  \mathcal{M}'_0 \preceq \mathcal{M}_0 \\
  \mathcal{M} = \mathcal{E}_0[\mathcal{M}_0] \mathcal{M}_2 \\
  \] 

  From the definition of $\preceq$

  By the induction hypothesis

  By the induction hypothesis

• **Case $\mathcal{E}' = W_1 \mathcal{E}_0$:**

  \[
  \mathcal{E}'[\mathcal{M}_0] \preceq M \\
  \mathcal{E}'[\mathcal{M}_0] \mathcal{M}_2 \preceq M \\
  \mathcal{M} = \mathcal{M}_1 \mathcal{M}_2 \\
  \mathcal{E}_0[\mathcal{M}_0] \mathcal{M}_2 \preceq \mathcal{M}_2 \\
  \mathcal{M}_2 = \mathcal{E}_0[\mathcal{M}_0] \\
  \mathcal{M}'_0 \preceq \mathcal{M}_0 \\
  \mathcal{M} = W_1 \mathcal{E}_0[\mathcal{M}_0] \\
  \] 

  From the definition of $\preceq$

  By the induction hypothesis

  By the induction hypothesis

  By above equations

\[ \square \]

**Lemma 58** (Evaluation contexts preserve precision).

If $\mathcal{E}'[\mathcal{M}_0] \preceq \mathcal{E}[\mathcal{M}_0]$ and $\mathcal{M}'_0 \preceq \mathcal{M}_0$ and $\mathcal{M}'_1 \preceq \mathcal{M}_1$ then $\mathcal{E}'[\mathcal{M}'_1] \preceq \mathcal{E}[\mathcal{M}_1]$.

**Proof.** By induction on the derivation of $\mathcal{E}'[\mathcal{M}'_0] \preceq \mathcal{E}[\mathcal{M}_0]$. All cases are straightforward, using the induction hypothesis and the definition of $\preceq$. \[ \square \]

**Lemma 59** (Reduction preserves precision).

If $\vdash \mathcal{M}_1 : T_1$ and $\vdash \mathcal{M}_1 : T_1$ and $\mathcal{M}_1 \preceq \mathcal{M}_1$ and $\mathcal{M}_1 \longrightarrow R \mathcal{M}_1'$ then either

(a) $\mathcal{M}_1$ is a value and $\mathcal{M}_1' \preceq \mathcal{M}_1$, or

(b) there exists $\mathcal{M}_2$ such that $\mathcal{M}_1 \longrightarrow R \mathcal{M}_2$ and $\mathcal{M}_2' \preceq \mathcal{M}_2$.

**Proof.** Proceed by case analysis on $\mathcal{M}_1' \longrightarrow R \mathcal{M}_1$.

- **Case $\phi_i' \leq \phi_1'$**

  \[
  \phi_i' \dashv \phi_1' \Rightarrow \text{ReduceUpcast} \\
  \mathcal{E}' \longrightarrow R \mathcal{E}' \\
  \mathcal{M}_1' \preceq \mathcal{M}_1 \\
  \] 

  Proceed by case analysis on $\mathcal{M}_1' \preceq \mathcal{M}_1$. \[2020/8/16\]
• Case  
\[ W' \ll M \quad (\phi_2 \ll \phi_1) \ll (\phi_2 \ll \phi_1) \]
\[ (\phi_2 \ll \phi_1)W' \ll (\phi_2 \ll \phi_1)M \]

By Lemma 55 (Precision on values), \( M = W \) as \( W' \ll M \). Since \( \phi_1 \ll \phi_2 \), it is the case that \( (\phi_2 \ll \phi_1) = \phi \).

Proceed by cases on the rule deriving \( \phi \).

- Case \( \text{Cast} \ll \text{Ref} \)  
  In this case, \( (\phi_2 \ll \phi_1) = \phi \). Since \( \phi_1 \ll \phi_2 \) by rule \( \text{ReduceUpcast} \), it follows that \( M_1 \rightarrow_R W \), and we already have \( M_2' \ll M_2 \).

- Case \( \text{Cast} \ll \text{E} \)  
  These rules do not have a safe cast on the left, so these cases are impossible.

- Case \( \text{Rule deriving (} +_i \ll +_i \text{) } \ll (\phi_2 \ll \phi_1) \)  
  In this case, \( (\phi_2 \ll \phi_1) = \phi \) and \( (\phi_2 \ll \phi_1) = (+_i \ll +) \).

  - \( \vdash (+_i \ll +)W' : T_1' \)
  - \( \vdash W' : T_1' +_i T_2' \)
  By inversion on rule \( \text{TCast} \).

  \[ W' = \text{inj}_i W_0' \]
  By Corollary 47.

  \[ W = \text{inj}_i W_0 \]
  By inversion on \( \text{inj}_i W_0' \ll W \).

  \[ M = \text{inj}_i W_0 \]
  By equality.

  \[ M_1 \rightarrow_R \text{inj}_i W_0 \]
  By \( \text{ReduceCastSuccess} \).

  \[ M_2' \ll M_2 \]
  By equality.

- Cases Remaining rules: 
  In the remaining rules, \( (\phi_2 \ll \phi_1) = \phi \). Thus, \( \phi_1 \ll \phi_2 \).

  By rule \( \text{ReduceUpcast} \), it follows that \( M_1 \rightarrow_R \text{inj}_i W_0 \) and it was already given that \( M_2' \ll M_2 \).

• Case  
\[ W' \ll M_1 \]
\[ (\phi_2 \ll \phi_1)W' \ll M_1 \]
\[ W' \ll M_1 \]
Subderivation

\( M_1 = W \)
By Lemma 55 (Precision on values)

\( M_2' \ll M_1 \)
By above equations

• Case  
\[ (+_i \ll +)\text{inj}_i W' \rightarrow_R \text{inj}_i W' \]
\[ M_1' \]
\[ M_2' \ll M_1 \]

Proceed by case analysis on \( M_1' \ll M_1 \).

• Case  
\[ \text{inj}_i W' \ll M \]
\[ (+_i \ll +) \ll (\phi_2 \ll \phi_1) \]
\[ (+_i \ll +)\text{inj}_i W' \ll (\phi_2 \ll \phi_1)M \]

Inversion on \text{inj}_i W' gives \( M = \text{inj}_i M_0 \) and \( W' \ll M_0 \).

By Lemma 55 (Precision on values), \( M_0 = W \).

Since \( (+_i \ll +) \) is a backward cast \( bc' \), to derive \( bc' \ll (\phi_2 \ll \phi_1) \), we either used \( \text{Cast} \ll \text{Ref} \) or \( \text{Cast} \ll \text{E} \). In the former case, we have \( (\phi_2 \ll \phi_1) = \phi \). By rule \( \text{ReduceCastSuccess} \), we have \( M_1 \rightarrow_R M \) and we already have \( M_2 \ll M_2 \).

In the latter case, we have \( (\phi_2 \ll \phi_1) = \phi \). By definition of being a safe cast, \( \phi_1 \ll \phi_2 \). Therefore, by rule \( \text{ReduceUpcast} \), we have \( M_1 \rightarrow_R M \) and we already have \( M_2 \ll M_2 \).

• Case  
\[ \text{inj}_i W' \ll M_1 \]
\[ (+_i \ll +)\text{inj}_i W' \ll M_1 \]
\[ \text{inj}_i W' \ll M_1 \]
Subderivation

\( M_1 = W \)
By Lemma 55 (Precision on values)

\( M_2' \ll M_1 \)
By above equations

• Case  
\[ \phi' = (+_i +) \quad i \neq k \]
\[ (+_i \ll \phi)\text{inj}_i W' \rightarrow_R \text{matchfail} \]
\[ M_1' \]
\[ M_2' \ll M_1 \]

Proceed by case analysis on \( M_1' \ll M_1 \).
• Case \( \text{inj}_i W' \preceq M \quad (\phi_k \leftarrow \phi') \preceq (\phi_2 \leftarrow \phi_1) \)

\[
\begin{array}{c}
(\phi_k \leftarrow \phi') \text{inj}_i W' \preceq (\phi_2 \leftarrow \phi_1)M \\
\text{inj}_i \text{matchfail} \quad M_1 \end{array}
\]

Since \( \vdash M_1 : T_i \) and \( M_1 \) is not a value nor is it matchfail, by Theorem 7 there exists \( M_2 \) such that \( M_1 \mapsto M_2 \). By definition, \( M_1 \text{matchfail} \preceq M_2 \).

• Case \( \text{inj}_i W' \preceq M_1 \)

\[
\begin{array}{c}
\text{inj}_i W' \preceq M_1 \\
\text{subderivation} \\
\text{w.r.} \quad M_1 = W \quad \text{By Lemma 55 (Precision on values)} \\
\text{w.r.} \quad M_1' \preceq M_1 \quad \text{By definition of} \preceq
\end{array}
\]

• Case

\[
\begin{array}{c}
\text{case(inj}_i W', \text{inj}_i x.M_i') \mapsto_R [W'/x]M_i' \\
\text{M}_1 \mapsto_R M_1' \quad \text{ReduceCase1} \\
\end{array}
\]

Proceed by inversion on \( \text{case(inj}_i W', \text{inj}_i x.M_i') \preceq M_1 \).

In the first case, \( M_1 = \text{case(M}_i, \text{inj}_i x.M_i) \):

\[
\begin{array}{c}
\text{inj}_i W' \preceq M \\
\text{M}_i' \preceq M_1 \\
M = \text{inj}_i M_0 \\
\text{From definition of} \preceq \\
W' \preceq M_0 \\
\text{From definition of} \preceq \\
M_0 = W \\
\text{By Lemma 55 (Precision on values)} \\
W' \preceq W \\
\text{By above equations} \\
M_1 = \text{case(inj}_i W_i, \text{inj}_i x.M_i') \\
\text{By above equations} \\
M_1 \mapsto_R [W'/x]M_i \\
\text{By rule ReduceCase1} \\
\text{w.r.} \quad [W'/x]M_i' \preceq [W'/x]M_i \\
\text{By Lemma 56 (Substitution preserves precision)}
\end{array}
\]

In the second case, \( M_1 = \text{case(M}_i, \text{inj}_i x_1.M_{i1}, \text{inj}_i x_2.M_{i2}) \):

\[
\begin{array}{c}
\text{inj}_i W' \preceq M \\
\text{M}_i' \preceq M_{i1} \\
M = \text{inj}_i M_0 \\
\text{From definition of} \preceq \\
W' \preceq M_0 \\
\text{From definition of} \preceq \\
M_0 = W \\
\text{By Lemma 55 (Precision on values)} \\
W' \preceq W \\
\text{By above equations} \\
M_1 = \text{case(inj}_i W_i, \text{inj}_i x_1.M_{i1}, \text{inj}_i x_2.M_{i2}) \\
\text{By above equations} \\
M_1 \mapsto_R [W'/x]M_{i1} \\
\text{By rule ReduceCase2} \\
\text{w.r.} \quad [W'/x]M_i' \preceq [W'/x]M_{i1} \\
\text{By Lemma 56 (Substitution preserves precision)}
\end{array}
\]

• Case

\[
\begin{array}{c}
\text{case(inj}_i W', \text{inj}_i x_1.M_{i1} M_{i2}^{inj}_2, \text{inj}_i x_2.M_{i2}^{inj}_2) \mapsto_R [W'/x]M_i' \\
\text{M}_1' \preceq M_1 \\
M_1 = \text{case(M}_i, \text{inj}_i x_1.M_{i1}, \text{inj}_i x_2.M_{i2}) \\
\text{From definition of} \preceq \\
inj_i W' \preceq M \\
\text{M}_i' \preceq M_{i1} \\
\text{Given} \\
M_{i1} = \text{inj}_i M_0 \\
\text{From definition of} \preceq \\
W' \preceq M_0 \\
\text{By Lemma 55 (Precision on values)} \\
M_0 = W \\
\text{By above equations} \\
W' \preceq W
\end{array}
\]

43 2020/8/16
\[ M_1 = \text{case}(\text{inj}_j, W, \text{inj}_j x_1. M_{11}, \text{inj}_j x_2. M_{21}) \] By above equations
\[ M_1 \mapsto_R [W/x_1]M_{11} \]

By rule \textit{ReduceCase2}
\[ [W'/x_1]M_{11} \ll [W/x_1]M_{11} \]

By Lemma \textit{56} (Substitution preserves precision)

- **Case**

\[ (\lambda x. M'_0) W' \mapsto_R [W'/x]M'_0 \]
\[ M'_1 \]
\[ M'_2 \]

By above equations
\[ M_1 = (\lambda x. M_0) W \]
\[ (\lambda x. M'_0) W \mapsto_R [W/x]M_0 \]

By rule \textit{Reduceβ}
\[ [W'/x]M_0 \ll [W/x]M \]

By Lemma \textit{56} (Substitution preserves precision)

\[ \textbf{Theorem 12} \] (Stepping preserves precision).
If \( \vdash M'_1 : T'_1 \) and \( \vdash M_1 : T_1 \) and \( M'_1 \ll M_1 \) and \( M'_1 \mapsto M'_2 \) then either
(a) \( M_1 \) is a value and \( M'_2 \ll M_1 \), or
(b) there exists \( M_2 \) such that \( M_1 \mapsto M_2 \) and \( M'_2 \ll M_2 \), or
(c) \( M_1 = \text{matchfail} \) and \( M'_2 \ll M_1 \).

\[ \textbf{Proof:} \] Proceed by case analysis on \( M'_1 \mapsto M'_2 \).

- **Case**

\[ \epsilon'([M'_0]_{11}) \mapsto \epsilon'([M'_0]_{22}) \]
\[ M'_1 \]
\[ M'_2 \]

By Lemma \textit{57} (Precision inversion on evaluation contexts)
\[ M_1 = \epsilon[M_0] \]
\[ M'_0 \ll M_0 \]

\( \vdash \epsilon'([M'_0]_{11}) : T'_1 \)

\( \vdash \epsilon'[M_0]_{11} : T_1 \)

\( \vdash M'_0 : T'_0 \)

\( \vdash M_0 : T_0 \)

\( M'_0 \mapsto_R \epsilon[M_0] \)

Proceed by case analysis on the result of applying Lemma \textit{59} (Reduction preserves precision)

In the first case, \( M_0 \) is a value and \( M'_0 \ll M_0 \). Since \( M'_0 \ll M_0 \) and \( \epsilon'([M'_0]_{11}) \ll \epsilon[M_0] \), by Lemma \textit{58} (Evaluation contexts preserve precision) it follows that \( \epsilon'([M'_0]_{22}) \ll \epsilon[M_0] \). This is alternative (a).

In the second case, \( M_0 \mapsto_R \epsilon[M_0] \) and \( \epsilon'([M'_0]_{22}) \ll \epsilon[M_0] \). Therefore, by rule \textit{StepContext}, it follows \( M_1 \mapsto \epsilon[M_0] \). Since \( M'_0 \ll M_0 \) and \( \epsilon'([M'_0]_{11}) \ll \epsilon[M_0] \), by Lemma \textit{58} (Evaluation contexts preserve precision) it follows that \( \epsilon'([M'_0]_{22}) \ll \epsilon[M_0] \). This is alternative (b).

- **Case**

\[ \epsilon' \neq [] \]

\( \epsilon'[\text{matchfail}] \mapsto \text{matchfail} \)

Since \( \vdash M_1 : T_1 \), by Theorem \textit{12} it follows that either \( M_1 \) is a value, or there exists \( M_2 \) such that \( M_1 \mapsto M_2 \), or \( M_1 = \text{matchfail} \).

In the first case, \( M'_2 = \text{matchfail} \ll M_1 \) by definition of \( \ll \), which is alternative (a).

In the second case, \( M'_2 = \text{matchfail} \ll M_2 \) by definition of \( \ll \), which is alternative (b).

In the first case, \( M'_2 = \text{matchfail} \ll \text{matchfail} = M_1 \) by definition of \( \ll \), which is alternative (c).

\[ \textbf{Theorem 13} \] (\( \ll \) respects convergence).
If \( M' \ll M \) where \( \vdash M' : T' \) and \( \vdash M : T \)
and \( M' \) converges then \( M \) also converges.
**Proof.** It is given that \( M' \) converges. By Definition 43, there exists a value \( W' \) such that \( M' \rightarrow^* W' \). Proceed by induction on the number of steps in \( M' \rightarrow^* W' \).

If \( M' = W' \) then \( W' \equiv M \). By Lemma 55 [Precision on values], \( M = W \) for some value \( W \). Therefore, \( M \) converges as well.

Otherwise, \( M' \) takes at least one step, that is, \( M' \rightarrow M'' \rightarrow^* W' \). Then \( M'' \) must also converge, with \( M'' \rightarrow^* W' \) in fewer steps than \( M' \rightarrow^* W' \). Since \( M' \rightarrow M'' \), proceed by case analysis on the result of applying Theorem 12.

- In the first case (a), \( M \) is a value, so \( M \) converges.
- In the second case (b), there exists \( M_0 \) such that \( M \leftarrow M_0 \) and \( M_0 \nRightarrow M_0 \).
  By Theorem 6, \( \vdash M_0 : T \) and similarly \( \vdash M_0 : T \).
  By the induction hypothesis, \( M_0 \) converges. Since \( M_0 \) converges, \( M \) must also converge to the same value.
- In the third case (c), \( M = \text{matchfail} \) and \( M_0 \nRightarrow M \).
  By inversion on \( M_0 \nRightarrow \text{matchfail} \) it follows that \( M_0 = \text{matchfail} \). But we know that \( M_0 \) converges, a contradiction. Hence, this case is impossible.

### D.6 Translation

**D.6.1 Soundness**

**Theorem 16** (Sum Translation soundness).

Given \( \delta' \) and \( \delta \), there exists \( C \) such that \( \delta' \Rightarrow \delta \Leftrightarrow \lnot \lnot \).

Moreover, if \( \Theta \vdash M : (T_1 | \delta'| T_2) \) then \( \Theta \vdash C[M] : (T_1 | \delta | T_2) \).

**Proof.** Proceed by case analysis on whether \( |\delta'| \leq |\delta| \).

- **Case** \( |\delta'| \leq |\delta| \):
  - \( T_1 \leq T_1 \) \quad By Lemma 43 [Reflexivity of subtyping]
  - \( T_2 \leq T_2 \) \quad By Lemma 43 [Reflexivity of subtyping]
  - \( |\delta'| \leq |\delta| \) \quad Given
  - \( (T_1 | \delta'| T_2) \leq (T_1 | \delta | T_2) \) \quad By definition of \( \leq \)
  - \( \delta' \Rightarrow \delta \Rightarrow \lnot \) \quad By rule CoeSub
  - \( \Theta \vdash M : (T_1 | \delta'| T_2) \) \quad Suppose
  - \( \Theta \vdash M : (T_1 | \delta | T_2) \) \quad By rule TSub
  - \( C[M] = M \) \quad By definition
  - \( \Theta \vdash C[M] : (T_1 | \delta | T_2) \) \quad By above equations

- **Case** \( |\delta'| \leq |\delta| \):
  - \( |\delta'| \leq |\delta| \) \quad Given
  - \( |\delta'| \Rightarrow |\delta| \Rightarrow (|\delta| \Leftrightarrow |\delta'|) \) \quad By rule CoeCast
  - \( \Theta \vdash M : (T_1 | \delta'| T_2) \) \quad Suppose
  - \( \Theta \vdash (|\delta| \Leftrightarrow |\delta'|)M : (T_1 | \delta | T_2) \) \quad By rule TCast
  - \( C[M] = (|\delta| \Leftrightarrow |\delta'|)M \) \quad By definition
  - \( \Theta \vdash C[M] : (T_1 | \delta | T_2) \) \quad By above equations

**Theorem 17** (Type translation soundness).

If \( A' \Rightarrow A \) then there exists \( C \) such that \( A' \Rightarrow A \Leftrightarrow C \).

Moreover, if \( \Theta \vdash M : [A'] \) then \( \Theta \vdash C[M] : [A] \).

**Proof.** By induction on the structure of the derivation of \( A' \Rightarrow A \).

- **Case** Unit \( \Rightarrow Unit \) Unit:
  - Unit \Rightarrow Unit \Rightarrow \lnot \lnot \) \quad By rule CoeUnit
  - \( \Theta \vdash M : [Unit] \) \quad Suppose
  - \( \Theta \vdash C[M] : [Unit] \) \quad By definition of \( C \)

- **Case** \( A' \Rightarrow A_1 \Rightarrow A_2 \Rightarrow A \)
  \( (A' | \delta'| A_2) \Rightarrow (A_1 | \delta A_2) \Rightarrow A \)
  Proceed by case analysis on the definition of \( \delta' \).
  In the first case, suppose \( \delta' \in \{+, +1\} \).
In the last case, suppose $\delta' \in \{+, 1, +_2\}$. Symmetric to the previous case, hence omitted.

In the second case, suppose $\delta' \in \{+_2, 1, 2, 1\}$. By definition of $\leq$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+, _2\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.

Next, for $\delta' \in \{1, 2\}$, suppose $\delta' \neq 1$ and $\delta' \neq 2$.

Next, for $\delta' \in \{+_2, 1, 2, 1\}$, suppose $\delta' \neq +$.
Lemma 60 (Cast insertion preserves precision).

If \( \delta'_1 \Rightarrow \delta'_2 \Rightarrow C' \) and \( \delta_1 \Rightarrow \delta_2 \Rightarrow C \)
and \( \delta'_1 \equiv \delta_1 \) and \( \delta'_2 \equiv \delta_2 \) and \( M' \equiv M \)
then \( C'[M'] \equiv C[M] \).

D.6.2 Precision

Theorem 11 depends on Lemma 60 (Cast insertion preserves precision), which uses a modified version of the translation that always inserts casts, even safe ones. In effect, the modified translation does not have rule \( \text{CoeSub} \) and always uses rule \( \text{CoeCast} \) (Figure 12). It also inserts safe casts \( C'_1 \) and \( C'_2 \), similar to \( \text{CoeCase2} \) in rules \( \text{CoeCase1L} \) and \( \text{CoeCase1R} \). See Figure 21.

Theorem 9 (Translation soundness).
If \( \Gamma \vdash e : A \) then there exists \( M \) such that \( \Gamma \vdash e : A \) and \( \Gamma \vdash M : \text{\textbf{|}} A \).

Proof. By induction on the structure of the derivation of \( \Gamma \vdash e : A \).

- **Case SVar**: Apply rules \( \text{STVar} \) and \( \text{TVar} \).
- **Case**: By rule \( \text{SVar} \).
Coercion $C$ coerces sum $|\delta'|$ to sum $|\delta|

$\delta' \Rightarrow \delta \leadsto C$

Figure 21. Part of the type-directed translation, modified to insert safe casts; differences highlighted

Proof. Note the following reasons for arriving at the result.

(a) If the translated sums are equal, that is, $|\delta'_1| = |\delta_1|$ and $|\delta'_2| = |\delta_2|$, we have $C' = C$. (Casts are unique; in this context, this is immediate because we are using a translation that generates casts even if they are safe, so there is only one rule, $\text{CoeCast}$, that derives the judgment.) Then the result follows from $M' \equiv M$ and the definition of $\equiv$.

(b) If $C' = (|\delta'_1| \equiv |\delta_1|)$ and $C = (|\delta_2| \equiv |\delta_1|)$ and $|\delta'_2| \equiv |\delta_1|$ then $C'[M'] \equiv C[M]$ by definition of $\equiv$ as $M' \equiv M$.

Proced by case analysis on $|\delta_1| \equiv \delta_1$ based on the reflexive, transitive closure of precision on sums.

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-1}, +1\} \Rightarrow \delta \leadsto C'_1$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase1L
    
    $\delta' \in \{+^{-2}, +2\} \Rightarrow \delta \leadsto C'_2$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase1R

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-1}, +1\} \Rightarrow \delta \leadsto C'_1$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-2}, +2\} \Rightarrow \delta \leadsto C'_2$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-3}, +3\} \Rightarrow \delta \leadsto C'_3$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-4}, +4\} \Rightarrow \delta \leadsto C'_4$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-5}, +5\} \Rightarrow \delta \leadsto C'_5$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-6}, +6\} \Rightarrow \delta \leadsto C'_6$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-7}, +7\} \Rightarrow \delta \leadsto C'_7$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-8}, +8\} \Rightarrow \delta \leadsto C'_8$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-9}, +9\} \Rightarrow \delta \leadsto C'_9$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2

- **Cases $+ \in (+, +)$.**
  - $\delta' \in \{+^{-10}, +10\} \Rightarrow \delta \leadsto C'_{10}$
    - $\delta' \Rightarrow \delta \leadsto C_3$ *CoeCase2
Lemma 61 (Coercion preserves precision). 
If $A'_1 \Rightarrow A'_2 \leftarrow C'$ and $A_1 \Rightarrow A_2 \leftarrow C$
and $A'_1 \subseteq A_1$ and $A'_2 \subseteq A_2$ and $M' \approx M$
then $C'[M'] \approx C[M]$.

Proof. By induction on the structure of the derivation of $A'_1 \Rightarrow A'_2 \leftarrow C'$.

- Case
  \[
  \begin{array}{l}
  \text{Unit } \Rightarrow \text{Unit } \leftarrow [] \quad \text{CoeUnit} \\
  \text{Unit } \subseteq A_1 \quad \text{Given}
  \\
  \text{Unit } \subseteq A_2 \quad \text{Given}
  \\
  A_1 = \text{Unit} \quad \text{By Lemma 61 (Precision inversion)}
  \\
  A_2 = \text{Unit} \quad \text{By Lemma 61 (Precision inversion)}
  \\
  \text{Unit } \Rightarrow \text{Unit } \leftarrow C \quad \text{Given}
  \\
  C = [] \quad \text{By inversion on CoeUnit}
  \\
  M' \approx M \quad \text{Given}
  \\
  C'[M'] \approx C[M] \quad \text{By definition of } C' \text{ and } C
  \end{array}
  \]

- Case
  \[
  \begin{array}{l}
  A'_1 \Rightarrow A'_2 \leftarrow C' \quad A'_2 \Rightarrow A'_2 \leftarrow C' \\
  (A_1 \Rightarrow A_1), (A_1 \Rightarrow A_2) \leftarrow \lambda x. C_1[x] \quad \text{Coe} \rightarrow
  \\
  A'_1 \rightarrow A'_2 \subseteq A_1 \quad \text{Given}
  \\
  A_1 = A_1 \rightarrow A_2 \quad \text{By Lemma 61 (Precision inversion)}
  \\
  A'_1 \subseteq A_1 \quad "
  \\
  A'_2 \subseteq A_2 \quad "
  \\
  A'_2 \rightarrow A'_2 \subseteq A_2 \quad \text{Given}
  \\
  A_2 = A_2 \rightarrow A_2 \quad \text{By Lemma 61 (Precision inversion)}
  \\
  A'_2 \subseteq A_2 \quad "
  \\
  A'_2 \subseteq A_2 \quad "
  \\
  (A_1 \Rightarrow A_1) \Rightarrow (A_1 \Rightarrow A_2) \leftarrow C \quad \text{Given}
  \\
  A_1 \Rightarrow A_1 \leftarrow C_1 \quad \text{By inversion on Coe} \rightarrow
  \\
  A_1 \Rightarrow A_2 \leftarrow C_2 \quad "
  \\
  C = \lambda x. C_2[[] C_1[x]] \quad "
  \\
  x \ll x \quad \text{By definition of } \ll
  \\
  A'_2 \Rightarrow A'_1 \leftarrow C'_1 \quad \text{Subderivation}
  \\
  C'_1[x] \ll C_1[x] \quad \text{By the induction hypothesis}
  \\
  M' \ll M \quad \text{Given}
  \\
  M'[C'_1[x]] \ll M[C_1[x]] \quad \text{By definition of } \ll
  \\
  A'_2 \Rightarrow A'_2 \leftarrow C'_2 \quad \text{Subderivation}
  \\
  C'_2[M'[C'_1[x]]] \ll C_2[M[C_1[x]]] \quad \text{By the induction hypothesis}
  \\
  \lambda x. C_2[M'[C'_1[x]]] \ll \lambda x. C_2[M[C_1[x]]] \quad \text{By definition of } \ll
  \end{array}
  \]

- Case
  \[
  \begin{array}{l}
  \delta'_1 \in \{+1, +1\} \quad +1 \Rightarrow \delta'_1 \leftarrow C'_1 \\
  A'_1 \Rightarrow A'_2 \leftarrow C'_1 \\
  \delta'_2 \Rightarrow C'_3 \\
  (A'_1 \delta'_1 A'_2) \Rightarrow (A'_1 \delta'_2 A'_2) \leftarrow C'_3 \quad \text{CoeCaseLL}
  \\
  \end{array}
  \]
Case 1: Consider the case when \( A_{12} \delta A_{22} \subseteq A_2 \) \( \text{Given} \)

\[
A_2 = A_{12} \delta A_{22} \quad \text{By Lemma 3 (Precision inversion)}
\]

\[
A_{12} \subseteq A_{12} \quad " \\
A_{22} \subseteq A_{22} \quad " \\
\delta_2 \subseteq \delta_2 \quad "
\]

Case 2: Consider the case when \( A_1 = A_{11} \delta A_{21} \) \( \text{Given} \)

\[
A_1 = A_{11} \delta A_{21} \quad \text{By Lemma 3 (Precision inversion)}
\]

\[
A_{11} \subseteq A_{11} \quad " \\
A_{21} \subseteq A_{21} \quad " \\
\delta_1 \subseteq \delta_1 \quad "
\]

Since \( \delta_1' \in \{+, +1, +2\} \) and \( \delta_1 \subseteq \delta_1 \), by definition of \( \subseteq \) it follows that \( \delta_1 \in \{+, +1, +2\} \) as well.

Consider the case when \( \delta_1 \in \{+, +1, +2\} \).

\[
(A_{11} \delta A_{21}) \Rightarrow (A_{12} \delta A_{22}) \leftarrow C \quad \text{Given}
\]

\[
\begin{align*}
A_{11} &\Rightarrow A_{12} \leftarrow C' \\
\delta_1' &\Rightarrow \delta_1 \leftarrow C_3 \\
C &\subseteq C_3[\text{case}(\ldots, \text{inj}_1 x_1, C_1 [x_1])]
\end{align*}
\]

\[
\begin{align*}
&\delta_1' \Rightarrow \delta_1 \leftarrow C_3 \\
&\text{By definition of } \subseteq \\
&\text{Subderivation}
\end{align*}
\]

Consider the case when \( \delta_1 \in \{+, +1, +2\} \).

\[
(A_{11} \delta A_{21}) \Rightarrow (A_{12} \delta A_{22}) \leftarrow C \quad \text{Given}
\]

\[
\begin{align*}
A_{11} &\Rightarrow A_{12} \leftarrow C_1 \\
A_{21} &\Rightarrow A_{22} \leftarrow C_2 \\
\delta_1 &\Rightarrow \delta_1 \leftarrow C_3 \\
\delta_1' &\Rightarrow \delta_1 \leftarrow C_1 \\
\text{By Lemma 50 (Cast insertion preserves precision)}
\end{align*}
\]

\[
\begin{align*}
&\delta_1' \Rightarrow \delta_1 \leftarrow C_3 \\
&\text{By definition of } \subseteq \\
&\text{Subderivation}
\end{align*}
\]

\* Case \text{CoeCase1R} \quad \text{Symmetric to the CoeCase1L case.}
Case

By induction on the structure of the derivation of \( \Gamma \).

Suppose \((\text{Translation preserves precision})\)

\[ (A_1 \delta_1 A_2) \Rightarrow (A_2 \delta_2 A_2) \Rightarrow C \]

Given

\[ A_1 \delta_1 A_2 \subseteq A_2 \]

\[ A_2 = A_1 \delta_1 A_2 \quad \text{By Lemma 4 (Precision inversion)} \]

\[ A_1 \delta_1 A_2 \subseteq A_1 \]

\[ A_1 = A_1 \delta_1 A_2 \quad \text{By Lemma 3 (Precision inversion)} \]

Since \( \delta_1 \in \{+, +, +, +\} \) and \( \delta_1 \subseteq \delta_1 \), by definition of \( \subseteq \) it follows that \( \delta_1 \in \{+, +, +, +\} \) as well.

\[ (A_1 \delta_1 A_2) \Rightarrow (A_1 \delta_1 A_2) \Rightarrow C \]

Given

\[ A_1 \delta_1 A_2 \subseteq A_2 \]

\[ A_1 = A_1 \delta_1 A_2 \quad \text{By the induction hypothesis} \]

\[ C = C_3 \text{[case(\{\}, \text{inj}_{1 x_1}, \text{inj}_{1 x_1}, \text{inj}_{2 x_2}, \text{inj}_{2 x_2}, \text{inj}_{2 x_2})]} \]

By definition of \( \subseteq \)

\[ M_1 \]

By Lemma 60 \((\text{Cast insertion preserves precision})\)

\[ \delta_1 \Rightarrow \delta_2 \Rightarrow C_3 \]

Subderivation

\[ C_3[M_3] \Rightarrow C_3[M_3] \quad \text{By Lemma 60 (Cast insertion preserves precision)} \]

\[ \text{Theorem 11 (Translation preserves precision).} \]

Suppose \( \Gamma' \subseteq \Gamma \) and \( e' \subseteq e \).

1. If \( \Gamma' \vdash e' \Leftrightarrow A' \) and \( \Gamma \vdash e \Leftrightarrow A \) then \( \Gamma' \vdash e' : A' \Leftrightarrow M' \) and \( \Gamma \vdash e : A \Leftrightarrow M \) where \( M' \leq M \).

2. If \( \Gamma' \vdash e' \rightarrow A' \) and \( \Gamma \vdash e \rightarrow A \) then \( \Gamma' \vdash e' : A' \rightarrow M' \) and \( \Gamma \vdash e : A \rightarrow M \) where \( A' \leq A \) and \( M' \leq M \).

Proof. By induction on the structure of the derivation of \( \Gamma' \vdash e' \Leftrightarrow A' \) (part 1) or \( \Gamma' \vdash e' \rightarrow A' \) (part 2).
• Case  \( \Gamma'(x) = A' \)
  \( \Gamma' \vdash x \Rightarrow A' \text{ SynVar} \)
  \( x \subseteq e \) Given
  \( e = x \) From definition of \( \subseteq \)
  \( \Gamma \vdash x \Leftarrow A \) Given
  \( \Gamma[x] = A \) By inversion on SynVar
  \( \Gamma'(x) = A' \) Premise
  \( \Gamma' \vdash x : A' \Leftarrow x \) By rule STVar
  \( \Gamma' \vdash x : A \Leftarrow x \) By rule STVar
  \( x \approx x \) By definition of \( \approx \)

• Case  \( \Gamma' \vdash e' \Rightarrow A'_0 \quad A'_0 \Rightarrow A' \quad \text{ChkSub} \)

  By inversion on \( \Gamma \vdash e \Leftarrow A \), rule ChkSub was applied.
  \( \Gamma \vdash e \Rightarrow A \) By inversion on ChkSub
  \( A_0 \Rightarrow A' \) Subderivation
  \( \Gamma' \vdash e' \Rightarrow A'_0 \) By the induction hypothesis
  \( \Gamma' \vdash e : A_0 \Leftarrow M_0 \) ”
  \( A'_0 \subseteq A_0 \) ”
  \( M'_0 \equiv M_0 \) ”
  \( A'_0 \Rightarrow A' \) By Lemma 17 (Directed consistency obeys Structural Equivalence)
  \( A_0 \Rightarrow A \) By Lemma 17 (Directed consistency obeys Structural Equivalence)
  \( A_0 \Rightarrow A' \Rightarrow C' \) By Theorem 17
  \( A_0 \Rightarrow A \Rightarrow C \) By Theorem 17
  \( A' \subseteq A \) Given
  \( C'[M'_0] \equiv C[M_0] \) By Lemma 61 (Coercion preserves precision)
  \( \Gamma' \vdash e' : A' \Leftarrow C'[M'_0] \) By rule STCSub
  \( \Gamma' \vdash e : A \Leftarrow C[M_0] \) By rule STCSub

• Case  \( \Gamma' \vdash e'_0 \Leftarrow A' \quad \text{SynAnno} \)

  \( (e'_0 :: A') \subseteq e \) Given
  \( e = (e_0 :: A) \) From definition of \( \subseteq \)
  \( e'_0 \subseteq e_0 \) ”
  \( A' \subseteq A \) ”
  \( \Gamma \vdash (e_0 :: A) \Rightarrow A \) Given
  \( \Gamma \vdash e_0 \Leftarrow A \) By inversion on rule SynAnno
  \( \Gamma' \subseteq \Gamma \) Given
  \( \Gamma' \vdash e'_0 \Leftarrow A' \) Subderivation
  \( \Gamma' \vdash e'_0 : A' \Leftarrow M' \) By the induction hypothesis
  \( \Gamma \vdash e_0 : A \Leftarrow M \) ”
  \( M' \subseteq M \) ”
  \( \Gamma' \vdash (e'_0 :: A') : A' \Leftarrow M' \) By rule STAnno
  \( \Gamma \vdash (e_0 :: A) : A \Leftarrow M \) By rule STAnno

• Case  \( \Gamma' \vdash () \Leftarrow \text{Unit} \text{ ChkUnitIntro} \)
\[ \begin{align*}
& (\cdot) \subseteq e \quad \text{Given} \\
& e = () \quad \text{From definition of } \subseteq \\
& \Gamma \vdash () \iff A \quad \text{Given} \\
& A = \text{Unit} \quad \text{By inversion on } \text{ChkUnitIntro} \\
\end{align*} \]

- **Case** \[ \begin{align*}
& \Gamma' \vdash (\cdot) : \text{Unit} \iff () \quad \text{By rule } \text{STUnitIntro} \\
& \Gamma' \vdash () : \text{Unit} \iff () \quad \text{By rule } \text{STUnitIntro} \\
& () \equiv () \quad \text{By definition of } \equiv
\end{align*} \]

- **Case** \[ \begin{align*}
& \Gamma' \vdash e' \equiv A' \quad \ast_2 \leq \delta' \\
& \Gamma' \vdash (\text{inj}_i e'_0) \equiv (A'_1 \delta' A'_2) \quad \text{ChkSumIntro}
\end{align*} \]

\[
\begin{align*}
\text{inj}_\ell e'_0 & \equiv e \quad \text{Given} \\
\text{From definition of } \subseteq \\
\text{inj}_\ell e'_0 & \equiv e_0 \\
\Gamma & \vdash (\text{inj}_\ell e_0) \equiv A \quad \text{By inversion on } \text{ChkSumIntro} \\
\Gamma & \vdash e_0 \equiv A_1 \\
A & = A_1 \delta A_2 \\
\ast_1' & \leq \delta
\end{align*} \]

\[
\begin{align*}
A'_1 \delta' A'_2 & \subseteq A_1 \delta A_2 \\
A'_1 & \subseteq A_1 \\
A'_2 & \subseteq A_2 \\
\ast_1' & \leq \ast_1' \\
A'_1 \ast_1' A'_2 & \subseteq A_1 \ast_1' A_2 \\
\text{By definition of } \subseteq
\end{align*} \]

\[
\begin{align*}
\Gamma' & \vdash e'_0 \equiv A'_1 \\
\Gamma' & \vdash e'_0 : A'_1 \rightarrow M'_0 \\
\Gamma & \vdash e_0 : A_1 \rightarrow M_0 \\
M'_0 & \equiv M_0
\end{align*} \]

\[
\begin{align*}
\Gamma' & \vdash (\text{inj}_\ell e'_0) : (A'_1 \ast_1' A'_2) \rightarrow (\text{inj}_\ell M'_0) \quad \text{By rule } \text{STSubIntro} \\
\Gamma' & \vdash (\text{inj}_\ell e_0) : (A_1 \ast_1' A_2) \rightarrow (\text{inj}_\ell M_0) \quad \text{By rule } \text{STSubIntro}
\end{align*} \]

\[
\begin{align*}
inj_\ell M'_0 & \equiv inj_\ell M_0 \quad \text{By definition of } \equiv
\end{align*} \]

\[
\begin{align*}
A'_1 & \leq A'_1 \\
A'_1 & \leq A_1 \\
A'_2 & \leq A'_2 \\
A'_2 & \leq A_2
\end{align*} \]

\[
\begin{align*}
A'_1 \ast_1' A'_2 & \leq A'_1 \delta' A'_2 \\
A'_1 \ast_1' A'_2 & \leq A_1 \delta A_2 \\
A'_1 \ast_1' A'_2 & \rightarrow C' \\
A'_1 \ast_1' A'_2 & \rightarrow A_1 \delta A_2 \rightarrow C
\end{align*} \]

\[
\begin{align*}
\text{By Lemma 2 (Relexivity of subtyping)} \\
\text{By Lemma 2 (Rellexivity of subtyping)} \\
\text{By Lemma 2 (Rellexivity of subtyping)} \\
\text{By Lemma 2 (Rellexivity of subtyping)} \\
\text{By definition of } \leq \\
\text{By definition of } \leq \\
\text{By Lemma 8 (Subtyping obeys directed consistency)} \\
\text{By Lemma 8 (Subtyping obeys directed consistency)} \\
\text{By Lemma 17 (Directed consistency obeys Structural Equivalence)} \\
\text{By Lemma 17 (Directed consistency obeys Structural Equivalence)} \\
\text{By Theorem 17} \\
\text{By Theorem 17}
\end{align*} \]

\[
\begin{align*}
\Gamma' & \vdash (\text{inj}_\ell e'_0) : (A'_1 \delta' A'_2) \rightarrow C'[\text{inj}_\ell M'_0] \\
\Gamma & \vdash (\text{inj}_\ell e_0) : (A_1 \delta A_2) \rightarrow C[\text{inj}_\ell M_0] \\
C'[\text{inj}_\ell M'_0] & \equiv C[\text{inj}_\ell M_0] \\
\text{By Lemma 61 (Coercion preserves precision)}
\end{align*} \]

- **Case** \[ \begin{align*}
& \Gamma' \vdash e'_0 \Rightarrow (A'_1 \delta' A'_2) \\
& \delta' \Rightarrow \ast_1' \\
& \Gamma', x : A'_1 \vdash e'_0 \equiv A' \\
\Gamma' & \vdash \text{case}(e'_0, \text{inj}_\ell x.e'_0) \equiv A'
\end{align*} \]

\[ \text{ChkSumElim1} \]
\[
\text{case}(e_0', \text{inj}_j x. e'_1) \subseteq e
\]
\[
e = \text{case}(e_0, \text{inj}_j x. e_1)
\]
\[
e_0' \subseteq e_0
\]
\[
e_1' \subseteq e_1
\]
\[
\Gamma \vdash \text{case}(e_0, \text{inj}_j x. e_1) \leftarrow A
\]
\[
\Gamma \vdash e_0 \Rightarrow (A_1 \delta A_2)
\]
\[
\Gamma, x : A_1 \vdash e_1' \leftarrow A
\]
\[
\delta \Rightarrow \ast
\]
\[
\Gamma' \subseteq \Gamma
\]
\[
\Gamma' \vdash e_0' \Rightarrow (A'_1 \delta' A'_2)
\]
Subderivation
\[
\Gamma' \vdash e_0' : (A'_1 \delta' A'_2) \hookrightarrow M_0'
\]
By the induction hypothesis
\[
A'_1 \delta' A'_2 \subseteq A_1 \delta A_2
\]
\[
M_0' \approx M_0
\]
\[
A'_1 \subseteq A_1
\]
\[
A'_2 \subseteq A_2
\]
From definition of \(\subseteq\)
\[
\Gamma', x : A_1 \vdash e_1' : A
\]
Subderivation
\[
\Gamma', x : A_1 \vdash e_1' : A \hookrightarrow M_i
\]
By the induction hypothesis
\[
M'_i \approx M_i
\]
Case \text{ChkSumElim2} Similar to the \text{ChkSumElim1} case, hence omitted.

Case \text{ChkSumElim1}
\[ \lambda x. e_0' \subseteq e \quad \text{Given} \]
\[ e = \lambda x. e_0 \quad \text{From definition of } \subseteq \]
\[ e_0' \subseteq e_0 \quad \text{"} \]
\[ \Gamma \vdash (\lambda x. e_2) \subseteq A \quad \text{Given} \]
\[ \Gamma, x : A_1 \vdash e_0 \subseteq A_2 \quad \text{By inversion on } \text{chk-intro} \]
\[ A = A_1 \rightarrow A_2 \quad \text{"} \]
\[ A_1' \rightarrow A_2' \subseteq A_1 \rightarrow A_2 \quad \text{Given} \]
\[ A_1' \subseteq A_1 \quad \text{From definition of } \subseteq \]
\[ A_2' \subseteq A_2 \quad \text{"} \]
\[ \Gamma' \subseteq \Gamma \quad \text{Given} \]
\[ \Gamma', x : A_1' \subseteq \Gamma, x : A_1 \quad \text{By definition of } \subseteq \]
\[ \Gamma', x : A_1' \vdash e_0' \subseteq A_2' \quad \text{Subderivation} \]
\[ \Gamma', x : A_1' \vdash e_0' : A_2' \rightarrow M_0' \quad \text{By the induction hypothesis} \]
\[ \Gamma, x : A_1 \vdash e_0 : A_2 \rightarrow M_0 \quad \text{"} \]
\[ M_0' \equiv M_0 \quad \text{"} \]
\[ \vdash \Gamma \vdash \langle \lambda x. e_0' \rangle : (A_1' \rightarrow A_2') \rightarrow (\lambda x. M_0') \quad \text{By rule } \text{ST-intro} \]
\[ \vdash \Gamma \vdash \langle \lambda x. e_0 \rangle : (A_1 \rightarrow A_2) \rightarrow (\lambda x. M_0) \quad \text{By rule } \text{ST-intro} \]
\[ \lambda x. M_0' \equiv \lambda x. M_0 \quad \text{By definition of } \equiv \]

• Case \[ \Gamma' \vdash e_1' \Rightarrow \langle A_0' \rightarrow A' \rangle \quad \Gamma' \vdash e_2' \subseteq A_0' \quad \text{Syn-elim} \]
\[ \Gamma' \vdash \langle e_1' e_2' \rangle \Rightarrow A' \quad \text{Syn-elim} \]
\[ e_1' e_2' \subseteq e \quad \text{Given} \]
\[ e = e_1 e_2 \quad \text{From definition of } \subseteq \]
\[ e_1' \subseteq e_1 \quad \text{"} \]
\[ e_2' \subseteq e_2 \quad \text{"} \]
\[ \Gamma \vdash e_1 e_2 \subseteq A \quad \text{By inversion on } \text{Syn-elim} \]
\[ \Gamma \vdash e_1 \Rightarrow \langle A_0 \rightarrow A \rangle \quad \text{"} \]
\[ \Gamma \vdash e_2 \subseteq A_0 \quad \text{"} \]
\[ \Gamma' \subseteq \Gamma \quad \text{Given} \]
\[ \Gamma' \vdash e_1' \Rightarrow \langle A_0' \rightarrow A' \rangle \quad \text{Subderivation} \]
\[ \Gamma' \vdash e_1' : (A_0' \rightarrow A') \rightarrow M_1' \quad \text{By the induction hypothesis} \]
\[ \Gamma \vdash e_1 : (A_0 \rightarrow A) \rightarrow M_1 \quad \text{"} \]
\[ A_0' \rightarrow A' \subseteq A_0 \rightarrow A \quad \text{"} \]
\[ M_1' \equiv M_1 \quad \text{"} \]
\[ \vdash \Gamma' \vdash \langle e_1' e_2' \rangle : A' \rightarrow \langle M_1' M_2' \rangle \quad \text{By rule } \text{ST-intro} \]
\[ \vdash \Gamma' \vdash \langle e_1 e_2 \rangle : A \rightarrow \langle M_1 M_2 \rangle \quad \text{By rule } \text{ST-intro} \]
\[ \vdash M_1' M_2' \equiv M_1 M_2 \quad \text{By definition of } \equiv \]

D.7 Static programs don’t go wrong

We write \( \Gamma_V \) for \( \Gamma \) restricted to the set of variables \( V \).

**Theorem 18** (Static programs don’t go wrong).

If \( \Gamma \vdash e : A \) by a static derivation then \( \Gamma_{\forall V(e)} \vdash e : A \rightarrow M \) and, for all \( M' \) such that \( M \rightarrow^* M' \), it is the case that \( M' \) free.

**Proof.** Apply Theorem 19 and Theorem 10 to show \( M \) free.

The result follows by induction on the number of steps in \( M \rightarrow^* M' \), using Theorem 8.

D.7.1 Static derivations

**Definition 2.** We say that a derivation of \( \Gamma \vdash e \subseteq A \) or \( \Gamma \vdash e \Rightarrow A \) is a static derivation if, for all subderivations deriving checking or synthesis judgments, the types checked or synthesized are static.
Note. If a derivation is static, then all of its subderivations must be static.

**Lemma 62** (Context thinning).
If \( y \notin FV(e) \) then:
1. If \( \Gamma, y : A' \vdash e \iff A \) then \( \Gamma \vdash e \iff A \).
2. If \( \Gamma, y : A' \vdash e \Rightarrow A \) then \( \Gamma \vdash e \Rightarrow A \).

**Proof.** By induction on the height of the given derivation.

- **Case** \( (\Gamma, y : A')(x) = A \)
  \[ \frac{\Gamma, y : A' \vdash x \Rightarrow A \quad y \neq x \quad \text{Since } y \notin FV(x) \quad (\Gamma, y : A')(x) = A}{\text{Premise}} \]
  \[ \Gamma \vdash x \Rightarrow A \text{ By definition} \]
  - **Case** \( \text{SynVar} \) Use the induction hypothesis and apply rule \( \text{SynVar} \)
  - **Case** \( \text{SynAnno} \) Use the induction hypothesis, and apply rule \( \text{SynAnno} \)
  - **Case** \( \text{ChkUnitIntro} \) Apply rule \( \text{ChkUnitIntro} \)
  - **Case** \( \text{ChkSumIntro} \) Use the induction hypothesis, the definition of \( FV(-) \), and apply rule \( \text{ChkSumIntro} \)
  - **Case** \( \text{ChkSumElim1} \) Use the induction hypothesis, the definition of \( FV(-) \), and apply rule \( \text{ChkSumElim1} \)
  - **Case** \( \text{ChkSumElim2} \) Use the induction hypothesis, the definition of \( FV(-) \), and apply rule \( \text{ChkSumElim2} \)
  - **Case** \( \text{ChkIntro} \) Use the induction hypothesis, the definition of \( FV(-) \), and apply rule \( \text{ChkIntro} \)
  - **Case** \( \text{Syn→Elim} \) Use the induction hypothesis, the definition of \( FV(-) \), and apply rule \( \text{Syn→Elim} \)

**Corollary 63** (Context support).
1. If \( \Gamma \vdash e \iff A \) then \( \Gamma |_{FV(e)} \vdash e \iff A \).
2. If \( \Gamma \vdash e \Rightarrow A \) then \( \Gamma |_{FV(e)} \vdash e \Rightarrow A \).

**Proof.** By induction on \( |\text{dom(}\Gamma\text{)} \setminus FV(e)| \).
If \( \text{dom}(\Gamma) = FV(e) \), then \( \Gamma = \Gamma |_{FV(e)} \) so we already have the result.
Otherwise, use the induction hypothesis, and apply Lemma 62 (Context thinning).

**Theorem 19** (Static subformula).
1. If \( \Gamma \vdash e \iff A \) by a static derivation then \( \Gamma^S \vdash e^S \iff A^S \) where \( \Gamma^S = \Gamma |_{FV(e)} \), \( e^S = e \), and \( A^S = A \).
2. If \( \Gamma \vdash e \Rightarrow A \) by a static derivation then \( \Gamma^S \vdash e^S \Rightarrow A^S \) where \( \Gamma^S = \Gamma |_{FV(e)} \), \( e^S = e \), and \( A^S = A \).

**Proof.** By induction on the height of the given derivation.
Since \( \Gamma \vdash e \iff A \) and \( \Gamma \vdash e \Rightarrow A \) by static derivations, all occurrences of types in checking and synthesizing positions are static, including \( A \). Therefore, \( A^S = A \) already holds.
Applying Corollary 63 individually to \( \Gamma \vdash e \iff A^S \) and \( \Gamma \vdash e \Rightarrow A^S \) produces the derivations \( \Gamma |_{FV(e)} \vdash e \iff A^S \) and \( \Gamma |_{FV(e)} \vdash e \Rightarrow A^S \) respectively.
Note that \( \Gamma |_{FV(e)} \vdash e \iff A^S \) and \( \Gamma |_{FV(e)} \vdash e \Rightarrow A^S \) are also static derivations.
All cases are then immediate by the induction hypothesis and applying the relevant rule.

---

**D.7.2 Static translations are free of casts and match failures**

**Notation.** We write \( M \text{ free} \) to denote that the target term \( M \) contains no casts or \( \text{matchfails} \).

**Lemma 64** (Subsums don’t need casts).
1. If \( i^* \leq \delta^S \) and \( i^* \Rightarrow \delta^S \iff C \) then \( C = [] \).
2. If \( i^* \leq \delta^S \) and \( i^* \Rightarrow \delta^S \iff C \) then \( C = [] \).

**Proof.**
1. From definition of subtyping, it is either the case that \( \delta^S = i^* \) or \( \delta^S = i^* \). In both cases, by definition of subtyping, \( i^* = i^* \leq \delta^S \). By inversion on \( i^* \Rightarrow \delta^S \iff C \), either rule \( \text{CoeSub} \) or \( \text{CoeCast} \) was applied. If rule \( \text{CoeCast} \) was applied then \( i^* \not\leq \delta^S \), a contradiction. If rule \( \text{CoeSub} \) was applied, then indeed \( C = [] \).
2. By definition of subtyping, \( i^* = i^* \leq i^* = i^* \). By inversion on \( i^* \Rightarrow i^* \iff C \), either rule \( \text{CoeSub} \) or \( \text{CoeCast} \) was applied. If rule \( \text{CoeCast} \) was applied then \( i^* \not\leq i^* \), a contradiction. If rule \( \text{CoeSub} \) was applied, then indeed \( C = [] \).

**Lemma 65** (Gradual sums in static don’t need casts).
1. If \( A_{11}^S +_1 A_{21}^S \leq A_{12}^S \delta^S A_{22}^S \) and \( A_{11}^S +_1 A_{21}^S \Rightarrow A_{12}^S \delta^S A_{22}^S \Rightarrow C \) and \( M \) free then \( C[M] \) free.

2. If \( A_{11}^S +_1 A_{21}^S \leq A_{12}^S +_1 A_{22}^S \) and \( A_{11}^S +_1 A_{21}^S \Rightarrow A_{12}^S +_1 A_{22}^S \Rightarrow C \) and \( M \) free then \( C[M] \) free.

**Proof:**

1. \( A_{11}^S +_1 A_{21}^S \Rightarrow A_{12}^S \delta^S A_{22}^S \Rightarrow C \) Given by inversion on CoeCase1L or CoeCase1R

\[
A_{12} \Rightarrow C_1
\]

\[
+_1 \Rightarrow \delta^S \Rightarrow C_2
\]

\[
C = C_3[\text{case}([], \text{inj}_1 x_1, \text{inj}_2 C_1[x_1])]
\]

2. Similar to the proof for the previous statement, hence omitted.

**Lemma 66** (Static sums don’t need casts).

If \( \delta^S \leq \delta^S \Rightarrow \delta^S \Rightarrow C \) then \( C = [] \).

**Proof.** By definition of sum translation, \( |\delta^S| = |\delta^S| \) and \( |\delta^S| = |\delta^S| \). Therefore, \( |\delta^S| \leq |\delta^S| \). By inversion on \( \delta^S \Rightarrow \delta^S \Rightarrow C \), either rule CoeSub or CoeCast was applied. If rule CoeCast was applied then \( |\delta^S| \leq |\delta^S| \), a contradiction. If rule CoeSub was applied, then indeed \( C = [] \).

**Lemma 67** (Static subtypes don’t need casts).

If \( A_1^S \leq A^S \) and \( A_1^S \Rightarrow A^S \Rightarrow C \) then \( C[M] \) free for any \( M \) free.

**Proof.** By induction on the structure of the derivation of \( A_1^S \Rightarrow A^S \Rightarrow C \).

- **Case** CoeUnit Immediate by the definition of \( C = [] \).

- **Case**

\[
A_{12}^S \Rightarrow A_{11}^S \Rightarrow C_1 \quad A_{21}^S \Rightarrow A_{22}^S \Rightarrow C_2
\]

\[
(A_{11}^S \Rightarrow A_{21}^S) \Rightarrow (A_{12}^S \Rightarrow A_{22}^S) \Rightarrow \lambda x. C_2[[]] C_1[x]
\]

\[
A_{11} \Rightarrow A_{21} \leq A_{12} \Rightarrow A_{22}
\]

\[
A_{11} \leq A_{12}
\]

\[
A_{21} \leq A_{22}
\]

\[
x \text{ free}
\]

\[
A_{12} \Rightarrow A_{11} \Rightarrow C_1
\]

\[
C_1[x] \text{ free}
\]

\[
M \text{ free}
\]

\[
M C_1[x] \text{ free}
\]

\[
A_{21} \Rightarrow A_{22} \Rightarrow C_2
\]

\[
C_2[M C_1[x]] \text{ free}
\]

\[
\lambda x. C_2[M C_1[x]] \text{ free}
\]

- **Case**

\[
A_{11}^S \Rightarrow A_{12}^S \Rightarrow C_1 \quad +_1 \Rightarrow \delta^S \Rightarrow C_3
\]

\[
(A_{11}^S +_1 A_{21}^S) \Rightarrow (A_{12}^S \delta^S A_{22}^S) \Rightarrow C_3[\text{case}([], \text{inj}_1 x_1, \text{inj}_1 C_1[x_1])]
\]
Proof. By induction on the structure of the given derivation.

- **Case** `SynVar` Apply rule `STVar` `M = x` is free of casts and matchfail.
- **Case** `CoeCase1R` Symmetric to the `CoeCase1L` case, hence omitted.

**Theorem 10** (Static derivations don’t have match failures).

If \( \Gamma \vdash e : A \) or \( \Gamma \vdash e : A \) then there exists \( M \) such that \( \Gamma \vdash e : A \) and \( M \) is free of casts and matchfail.

Proof. By induction on the structure of the given derivation.

1. **Case** `SynVar` Apply rule `STVar` `M = x` is free of casts and matchfail.
2. **Case** `CoeCase1R` Symmetric to the `CoeCase1L` case, hence omitted.

\[
\begin{align*}
A_1^S + A_2^S &\leq A_2^S \delta^S A_2^S \\
A_1^S &\leq A_2^S \\
A_2^S &\leq A_2^S \\
+ &\leq \delta^S \\
+ &\Rightarrow \delta^S \rightarrow C_3 \\
C_3 &\equiv [ ]
\end{align*}
\]

\[
\begin{align*}
A_1^S &\leq A_2^S \delta^S A_2^S \\
A_1^S &\leq A_2^S \\
A_2^S &\leq A_2^S \\
+ &\leq \delta^S \\
+ &\Rightarrow \delta^S \rightarrow C_3 \\
C_3 &\equiv [ ]
\end{align*}
\]

\[
\begin{align*}
A_1^S &\leq A_2^S \delta^S A_2^S \\
A_1^S &\leq A_2^S \\
A_2^S &\leq A_2^S \\
+ &\leq \delta^S \\
+ &\Rightarrow \delta^S \rightarrow C_3 \\
C_3 &\equiv [ ]
\end{align*}
\]

\[
\begin{align*}
\text{By Lemma 1 (Subtyping inversion)} \\
\text{By the induction hypothesis} \\
\text{By the definition of free} \\
\text{Subderivation} \\
\text{By the definition of free} \\
\text{Subderivation} \\
\text{Subderivation} \\
\text{By inversion on CoeSub} \\
\text{By inversion on CoeSub} \\
\text{By Lemma 66 (Static sums don’t need casts)}
\end{align*}
\]

- **Case** `CoeCase1R` Symmetric to the `CoeCase1L` case, hence omitted.
Case ChkUnitIntro

Apply rule STUnitIntro. M = () is free of casts and matchfail.

\[ \Gamma^S \vdash e^S : (A^S \delta^S A^S) \quad \delta^S \Rightarrow \star^+ \]

Subderivation

\[ \delta^S = \star^+ \]

By Lemma [33] Static looseness, II.

\[ A^S \delta^S A^S \leq A^S \delta^S A^S + \star^+ A^S \]

By definition of \( \leq \).

\[ A^S \delta^S A^S \leq A^S \delta^S A^S + \star^+ A^S \]

By Lemma [38] Subtyping obeys directed consistency.

\[ A^S \delta^S A^S \leq A^S \delta^S A^S + \star^+ A^S \]

By definition of \(<\).

\[ A^S \delta^S A^S \leq A^S \delta^S A^S + \star^+ A^S \]

By Lemma [38] Subtyping obeys directed consistency.

\[ A^S \delta^S A^S \leq A^S \delta^S A^S + \star^+ A^S \]


\[ A^S \delta^S A^S \Rightarrow A^S \delta^S A^S + \star^+ A^S \rightarrow C \]

By Theorem [17].

\[ \Gamma^S \vdash e^S : (A^S \delta^S A^S) \quad \delta^S \Rightarrow \star^+ \]

Subderivation

\[ \Gamma^S \vdash e^S : A^S \delta^S \leftarrow M_0 \quad \text{By the induction hypothesis} \]

\[ M_0 \text{ free} \]

By rule STSub.

\[ C[M_0] \text{ free} \]

By Lemma [65] Gradual sums in static don’t need casts.

\[ \Gamma^S, x : A^S \vdash e^S : A^S \]

Subderivation

\[ \Gamma^S, x : A^S \vdash e^S : A^S \leftarrow M_1 \quad \text{By the induction hypothesis} \]

\[ M_1 \text{ free} \]

By rule STSumIntro.

\[ \Gamma^S, x : A^S \vdash e^S \vdash (C[M_0], \text{inj}, x, M_1) \quad \text{By rule STSumIntro} \]

\[ \text{case}(C[M_0], \text{inj}, x, M_1) \text{ free} \]

By definition of free.
• Case $\Gamma^S \vdash e^s_0 \Rightarrow (A^S_1 \delta^S A^S_2)$
  $\delta^S \Rightarrow +$

  \[
  \Gamma^S \vdash \text{case}(e^s_0, \text{inj}_1 x_1.e^s_1, \text{inj}_2 x_2.e^s_2) \Leftarrow A^S \quad \text{ChkSumElim2}
  \]

  \[A^S_1 \leq A^S_1 \quad \text{By Lemma 2 (Reflexivity of subtyping)}\]
  \[A^S_2 \leq A^S_2 \quad \text{By Lemma 2 (Reflexivity of subtyping)}\]
  \[\delta^S \leq + \quad \text{By Lemma 23 (All sums below +)}\]
  \[A^S_1 \delta^S A^S_2 \leq A^S_1 + A^S_2 \quad \text{By definition of $\leq$}\]
  \[A^S_1 \delta^S A^S_2 \bowtie A^S_1 + A^S_2 \quad \text{By Lemma 8 (Subtyping obeys directed consistency)}\]
  \[A^S_1 \delta^S A^S_2 \Rightarrow A^S_1 + A^S_2 \quad \text{By Theorem 17}\]

  \[
  \Gamma^S \vdash e^s_0 \Rightarrow (A^S_1 \delta^S A^S_2)
  \]
  Subderivation

  \[
  \Gamma^S \vdash e^s_0 : (A^S_1 \delta^S A^S_2) \Rightarrow M_0
  \]
  By the induction hypothesis

  \[
  M_0 \text{ free}
  \]
  By rule STSub

  \[
  \Gamma^S \vdash e^s_0 : (A^S_1 + A^S_2) \Rightarrow C[M_0]
  \]
  By Lemma 67 (Static subtypes don't need casts)

  \[
  \Gamma^S, x_1 : A^S_1 \vdash e^s_1 \Leftarrow A^S
  \]
  Subderivation

  \[
  \Gamma^S, x_1 : A^S_1 \vdash e^s_1 : A^S \Rightarrow M_1
  \]
  By the induction hypothesis

  \[
  M_1 \text{ free}
  \]
  By definition of free

  \[
  \Gamma^S, x_2 : A^S_2 \vdash e^s_2 \Leftarrow A^S
  \]
  Subderivation

  \[
  \Gamma^S, x_2 : A^S_2 \vdash e^s_2 : A^S \Rightarrow M_2
  \]
  By the induction hypothesis

  \[
  M_2 \text{ free}
  \]
  By definition of free

* Case Chk→Intro Use the induction hypothesis, the definition of free, and apply rule ST→Intro
* Case Syn→Elim Use the induction hypothesis, the definition of free, and apply rule ST→Elim