Abstract. Refinement types turn typechecking into lightweight verification. The classic form of refinement type is the datasort refinement, in which datasorts identify subclasses of inductive datatypes. Existing type systems for datasort refinements require that all the refinements of a type be specified when the type is declared; multiple refinements of the same type can be obtained only by duplicating type definitions, and consequently, duplicating code.

We enrich the traditional notion of a signature, which describes the inhabitants of datasorts, to allow re-refinement via signature extension, without duplicating definitions. Since arbitrary updates to a signature can invalidate the inversion principles used to check case expressions, we develop a definition of signature well-formedness that ensures that extensions maintain existing inversion principles. This definition allows different parts of a program to extend the same signature in different ways, without conflicting with each other. Each part can be type-checked independently, allowing separate compilation.

1 Introduction

Type systems provide guarantees about run-time behaviour; for example, that a record will not be multiplied by a string. However, the guarantees provided by traditional type systems like Hindley–Milner do not rule out a practically important class of run-time failures: nonexhaustive match exceptions. For example, the type system of Standard ML allows a case expression over lists that omits a branch for the empty list:

\[
\text{case } \text{elems} \text{ of } \text{head :: tail } \Rightarrow \text{head}
\]

If this expression is evaluated with \text{elems} bound to the empty list [], the exception \text{Match} will be raised.

Datasort refinements eliminate this problem: a datasort can express, within the static type system, that \text{elems} is not empty; therefore, the above case expression will never raise \text{Match}. Datasorts can also express less shallow properties. For example, the definition in Figure 1 encodes conjunctive normal form—a formula that consists of (possibly nested) \text{And}s of clauses, where a clause consists of (possibly nested) \text{Or}s of literals, where a literal is either a positive literal (a variable) or a negation of a positive literal. A case expression comparing two
values of type \texttt{clause} would only need branches for \texttt{Or}, \texttt{Not} and \texttt{Var}; the \texttt{And} branch could be omitted, since \texttt{And} does not produce a \texttt{clause}.

Datasorts correspond to regular tree grammars, which can encode various data structure invariants (such as the colour invariant of red-black trees), as well as properties such as CNF and A-normal form. Datasort refinements are less expressive than the “refinement type” systems (such as liquid types) that followed work on index refinements and indexed types; like regular expressions, which “can’t count”, datasorts cannot count the length of a list or the height of a tree. However, types with datasorts are simpler in some respects; most importantly, types with datasorts never require quantifiers. Avoiding quantifiers, especially existential quantifiers, also avoids many complications in type checking. By analogy, regular expressions cannot solve every problem—but when they can solve the problem, they may be the best solution.

The goal of this paper is to make datasort refinements more usable—not by making datasorts express more invariants, but by liberating them from the necessity of a fixed specification (a fixed signature). First, we review the trajectory of research on datasorts.

The first approach to datasort refinements \cite{Freeman91,Freeman94} extended ML, using abstract interpretation \cite{Cousot77} to infer refined types. The usual argument in favour of type inference is that it reduces a direct burden on the programmer. When type annotations are boring or self-evident, as they often are in plain ML, this argument is plausible. But datasorts can express more subtle specifications, calling that argument into question. Moreover, inference discourages a form of fine-grained modularity. Just as we expect a module system to support information hiding, so that clients of a module cannot depend on its internal details, a type system should prevent the callers of a function from depending on its internal details. Inferring refinements exposes those details. For example, if a function over lists is written with only nonempty input in mind, the programmer may not have thought about what the function should do for empty input, so the type system shouldn’t let the function be applied to an empty list. Finally, inferring all properties means that the inferred refined types can be long, e.g. inferring a 16-part intersection type for a simple function \cite[p. 271]{Freeman91}.

Thus, the second generation of work on datasort refinements \cite{Davies00,Davies05} used bidirectional typing, rather than inference. Programmers have to write more annotations, but refinement checking will never fabricate unintended invariants. A third generation of work \cite{Dunfield05}...
ning 2004; Dunfield 2007b) stuck with bidirectional type checking, though this was overdetermined: other features of that type system made inference untenable.

All three generations (and later work by Lovas 2010 on datasorts for LF) shared the constraint that a given datatype could be refined only once. The properties tracked by datasorts could not be subsequently extended; the same set of properties must be used throughout the program. Modular refinement checking could be achieved only by duplicating the type definition and all related code. Separate type-checking of refinements enables simpler reasoning about programs, separate compilation, and faster type-checking (simpler refinement relations lead to simpler case analyses).

The history of pattern typing in case expressions is also worth noting, as formulating pattern typing seems to be the most difficult step in the design of datasort type systems. Freeman supported a form of pattern matching that was oversimplified. Davies implemented the full SML pattern language and formalized most of it, but omitted as-patterns—which become nontrivial when datasort refinements enter the picture.

The system in this paper allows multiple, separately declared refinements of a type by revising a fundamental mechanism of datasort refinements: the signature. Refinements are traditionally described using a signature that specifies—for the entire program—which values of a datatype belong to which refinements. For example, the type system can track the parity of bitstrings using the following signature, which says: (1) even and odd are subsorts (subtypes) of the type bits of bitstrings, the (2) empty bitstring has even parity, (3) appending a 1 flips the parity, and (4) appending a 0 preserves parity.

\[
\begin{align*}
\text{even} \preceq \text{bits}, & \quad \text{odd} \preceq \text{bits}, \\
\text{Empty} : & \quad \text{even}, \\
\text{One} : & \quad (\text{even} \to \text{odd}) \land (\text{odd} \to \text{even}), \\
\text{Zero} : & \quad (\text{even} \to \text{even}) \land (\text{odd} \to \text{odd})
\end{align*}
\]

The connective \(\land\), read “and” or “intersection”, denotes conjunction of properties: adding a One makes an even bitstring odd \((\text{even} \to \text{odd})\), and makes an odd bitstring even \((\text{odd} \to \text{even})\). Thus, if \(b\) is a bitstring known to have odd parity, then appending a 1 yields a bitstring with even parity:

\[
b : \text{odd} \vdash \text{One}(b) : \text{even}
\]

In some datasort refinement systems (Dunfield 2007b; Lovas 2010), the programmer specifies the refinements by writing a signature like the one above. In the older systems of Freeman and Davies, the programmer writes a regular tree grammar from which the system infers a signature, including the constructor types and the subsort relation:

---

1 A regular tree grammar is like a regular grammar (the class of grammars equivalent to regular expressions), but over trees instead of strings (Comon et al. 2008); the leftmost terminal symbol in a production of a regular grammar corresponds to the symbol at the root of a tree.
\begin{align*}
even &= \text{Empty} \parallel \text{Zero}(\text{even}) \parallel \text{One}(\text{odd}) \\
\text{odd} &= \text{Zero}(\text{odd}) \parallel \text{One}(\text{even})
\end{align*}

In either design, the typing phase uses the same form of signature. We use the first design, where the programmer gives the signature directly. Giving the signature directly is more expressive, because it enables refinements to carry information not present at run time. For example, we can refine natural numbers by \text{Tainted} and \text{Untainted}:

\begin{align*}
Z & : \text{nat}, \\
S & : \text{nat} \to \text{nat}, \\
\text{tainted} & \preceq \text{nat}, \\
\text{untainted} & \preceq \text{nat}, \\
Z & : \text{tainted}, \\
S & : \text{tainted} \to \text{tainted}, \\
Z & : \text{untainted}, \\
S & : \text{untainted} \to \text{untainted}
\end{align*}

The sorts \text{tainted} and \text{untainted} have the same closed inhabitants, but a program cannot directly create an instance of \text{untainted} from an instance of \text{tainted}:

\[x : \text{tainted} \not\vdash S(x) : \text{untainted}\]

Thus, the two sorts have different \emph{open} inhabitants. This is analogous to dimension typing, where an underlying value is just an integer or float, but the type system tracks that the number is in (for example) metres (\cite{Kennedy1996}).

Giving the signature directly allows programmers to choose between a variety of subsorting relationships. For example, to allow untainted data to be used where tainted data is expected, write \text{untainted} \preceq \text{tainted}. Subsorting can be either \text{structural} (as the signatures generated from grammars) or \text{nominal} (as in the example above). In this paper, giving signatures directly is helpful: it enables extension of signatures without translating between signatures and grammars.

\textbf{Contributions.} This paper makes the following contributions:

\begin{itemize}
\item A language and type system with \textit{extensible signatures} for datasort refinements (Section 3). Refinements are extended by \textit{blocks} that are checked to ensure that they do not weaken a sort’s inversion principle, which would make typing unsound.
\item A new formulation of typing (Section 4) for case expressions. This formulation is based on a notion of finding the intersection of a type with a pattern; it concisely models the interesting aspects of realistic ML-style patterns.
\item Type (datasort) preservation and progress for the type assignment system, stated in Section 6 and proved in Appendix B with respect to a standard call-by-value operational semantics (Section 5).
\item A bidirectional type system (Section 7), which directly yields an algorithm. We prove that this system is sound (given a bidirectional typing derivation, erasing annotations yields a type assignment derivation) and complete (given any type assignment derivation, annotations can be added to make bidirectional typing succeed).
\end{itemize}

The appendix, which includes definitions and proofs omitted for space reasons, can be found at http://www.cs.queensu.ca/~jana/papers/extensible/.
2 Datasort Refinements

What are datasort refinements? Datasort refinements are a syntactic discipline for enforcing invariants. This is a play on Reynolds’s definition of types as a “syntactic discipline for enforcing levels of abstraction” [Reynolds 1983]. Datasorts allow programmers to conveniently categorize inductive data, and operations on such data, more precisely than in conventional type systems.

Indexed types and related systems (e.g. liquid types and other “refinement types”) also serve that purpose, but datasorts are highly syntactic, whereas indexed types depend on the semantics of a constraint domain. For example, to check the safety of accessing the element at position $2k$ of a 0-based array of length $n$, an indexed type system must check whether the proposition $2k < n$ is entailed in the theory of integers (under some set of assumptions, e.g. $0 \leq k \leq n/3$). The truth of $2k < n$ depends on the semantics of arithmetic, whereas membership in a datasort only depends on a head constructor and the datasorts of its arguments. Put roughly, datasorts express regular grammars, and indexed types express grammars with more powerful side conditions. (Unrestricted dependent types can express arbitrarily precise side conditions.)

Applications of datasort refinements. Datasorts are especially suited to applications of symbolic computing, such as compilers and theorem provers. Compilers usually work with multiple internal languages, from abstract syntax through to intermediate languages. These internal languages may be decomposed into further variants: source ASTs with and without syntactic sugar, A-normal form, and so on. Similarly, theorem provers, SMT solvers, and related tools transform formulas into various normal forms or sublanguages: quantifier-free Boolean formulas, conjunctive normal form, formulas with no free variables, etc. Many such invariants can be expressed by regular tree grammars, and hence by datasorts.

Our extensible refinements offer the ability to use new refinements of a datatype when the need arises, without the need to update a global refinement declaration. For example, we could extend the types in Figure 1 in which clause contains disjunctions of literals and cnf contains conjunctions of clauses, with a new sort for conjunctions of literals:

\[
\text{literal} \preceq \text{conj-literal}, \quad \text{conj-literal} \preceq \text{cnf},
\]

And:

\[ (\text{conj-literal} + \text{conj-literal}) \rightarrow \text{conj-literal} \]

What are datasort refinements not? First, datasorts are not really types, at least not in the sense of Hindley–Milner type systems. A function on bitstrings (Section 1) has a best, or principal, type: \text{bits} \rightarrow \text{bits}. In contrast, such a function may have many refined types (sometimes called sorts), depending not only on the way the programmer chose to refine the \text{bits} type, but on which possible properties they wish to check. The type, or sort, of a function is a tiny module interface. In a conventional Hindley–Milner type system, there is a best interface (the principal type); with datasorts, the “best” interface is—as with a module interface, which may reveal different aspects of the module—the one the programmer thinks best.
Term vars. $x, y, \ldots$

Expressions $e ::= x \mid \lambda x. e \mid e_1 e_2 \mid (e_1, e_2) \mid \text{c}(e) \mid \text{case } e \text{ of } ms$

| declare $\Sigma$ in $e$ — signature extension (Fig. 1)

Matches $ms ::= \cdot \mid ((p \Rightarrow e) \mid ms)

Values $v ::= x \mid \lambda x. e \mid (v_1, v_2) \mid \text{c}(v) \mid (v : A)$

Patterns $p ::= \_ \mid \emptyset \mid \text{c}(p) \mid (p_1, p_2) \mid x \text{ as } p \mid p_1 \sqcup p_2$

Fig. 2. Expressions

Datasorts $s, t, \ldots

Types $A, B, D ::= 1 \mid A \rightarrow B \mid A \ast B \mid s \mid A \land B$

Typing contexts $\Gamma ::= \cdot \mid \Gamma, x : A$

Fig. 3. Types and contexts

Maybe the programmer only cares that the function preserves odd parity, and annotates it with $\text{odd} \rightarrow \text{odd}$; the compiler will reject calls with even bitstrings, even though such a call would be conventionally well-typed.

To infer sorts, as in the original work of Freeman, is like assuming that all declarations in a module should be exposed. (Tools that suggest possible invariants could be useful, just as a tool that suggests possible module interfaces could be useful. But such tools are not the focus of this paper.)

3 A Type System with Extensible Refinements

This section gives our language’s syntax, introduces signatures, discusses the introduction and elimination forms for datasorts, and presents the typing rules. The details of typing pattern matching are in Section 4.

3.1 Syntax

The syntax of expressions (Figure 2) includes functions $\lambda x. e$, function application $e_1 e_2$, pairs $(e_1, e_2)$, constructors $\text{c}(e)$, and case expressions. Signatures are extended by declare $\Sigma$ in $e$.

Types (Figure 3), written $A$ and $B$, include unit ($1$), function, and product types, along with datasorts $s$ and $t$. The intersection type $A \land B$ represents the conjunction of the two properties denoted by $A$ and $B$; for example, a function to repeat a bitstring could be checked against type $(\text{odd} \rightarrow \text{even}) \land (\text{even} \rightarrow \text{even})$: given any bitstring $b$, the repetition $bb$ has even parity.

3.2 Unrefined types and signatures

Our unrefined types $\tau$, in Figure 4, are very simple: unit $1$, functions $\tau_1 \rightarrow \tau_2$, products $\tau_1 \ast \tau_2$, and datatypes $d$. We assume that each datatype has a known set of constructors: for example, the bitstring type of Section 7 has constructors
Unrefined datatype names \( d \)
Unrefined types \( \tau ::= 1 | \tau_1 \rightarrow \tau_2 | \tau_1 \ast \tau_2 | d \)
Unrefined signatures \( U ::= \cdot | U, d | U, c : \tau \rightarrow d \)

Constructor types \( C ::= A \rightarrow s \)
Blocks \( K ::= \cdot \) empty block
\( | K, s_1 \leq s_2 \) subsorting declaration
\( | K, c : C \) constructor type decl.
Sort sets \( S ::= (s_1 \sqsubseteq d_1, \ldots, s_n \sqsubseteq d_n) \)
Abbrev. sort sets \( S ::= (s_1, \ldots, s_n) \)
Signatures \( \Sigma, \Omega ::= \cdot \) empty signature
\( | \Sigma, S(K) \) datasort specification

\[
\Sigma \vdash A \sqsubseteq \tau
\]

Under signature \( \Sigma \) (and unrefined signature \( U \)), type \( A \) is a refinement of unrefined type \( \tau \)

\[
\Sigma \vdash 1 \sqsubseteq 1
\]
\[
\Sigma \vdash A_1 \sqsubseteq \tau_1 \quad \Sigma \vdash A_2 \sqsubseteq \tau_2 \quad \Rightarrow \quad \Sigma \vdash (A_1 \rightarrow A_2) \sqsubseteq (\tau_1 \rightarrow \tau_2)
\]
\[
\Sigma \vdash (A_1 \ast A_2) \sqsubseteq (\tau_1 \ast \tau_2)
\]

\[
(s \sqsubseteq d) \in \Sigma
\]
\[
\Sigma \vdash s \sqsubseteq d
\]
\[
\boxed{\Sigma \vdash A_1 \sqsubseteq \tau \quad \Sigma \vdash A_2 \sqsubseteq \tau \quad \Rightarrow \quad \Sigma \vdash (A_1 \land A_2) \sqsubseteq \tau}
\]

Fig. 4. Unrefined types and signatures, refined signatures, \( \sqsubseteq \)

Empty, One and Zero. Refinements don’t add constructors; they only refine the types of the given constructors. We assume that each program has some unrefined signature \( U \) that gives datatype names \( (d) \) and (unrefined) constructor typings \( (c : \tau \rightarrow d) \). Since this signature is the same throughout a program, we elide it in most judgment forms.

The judgment \( \Sigma \vdash A \sqsubseteq \tau \) says that \( A \) is a refinement of \( \tau \). Both the symbol \( \sqsubseteq \) and several of the rules are reminiscent of subtyping, but that is misleading: sorts and types are not in an inclusion relation in the sense of subtyping, because the rule for \( \rightarrow \) is covariant, not contravariant. Covariance is needed for functions whose domains are nontrivially refined, e.g. \( \text{odd} \rightarrow \cdots \), which is not a subtype of \( \text{bits} \rightarrow \cdots \) because \( \text{bits} \not\leq \text{odd} \).

Rule \( \sqsubseteq \land \) implements the usual refinement restriction: both parts of an intersection \( A_1 \land A_2 \) must refine the same unrefined type \( \tau \).

3.3 Signatures

Refinements are defined by signatures \( \Sigma \) (Figure 4).

As in past datasort systems, we separate signatures \( \Sigma \) from typing contexts \( \Gamma \). Typing assumptions over term variables \( (x, y, \text{etc.}) \) in \( \Gamma \) can mention sorts declared in \( \Sigma \), but the signature \( \Sigma \) cannot mention the term variables declared in \( \Gamma \). Thus, our judgment for term typing will have the form \( \Sigma; \Gamma \vdash e : A \), where the term \( e \) can include constructors declared in \( \Sigma \) and variables declared in \( \Gamma \), and the type \( A \) can include sorts declared in \( \Sigma \). Some judgments, like subsorting
\[
\Sigma \vdash A \text{ type} \quad \text{Type } A \text{ is well-formed}
\]

\[
\begin{array}{c}
\Sigma \vdash 1 \text{ type } \quad \text{WfType}\ast \\
\Sigma \vdash A_1 \text{ type} \\
\Sigma \vdash A_2 \text{ type} \\
\Sigma \vdash A_1 \to A_2 \text{ type} \quad \text{WfType}→ \\
\text{WfType} \ast \\
\Sigma \vdash A_1 \land A_2 \text{ type} \quad \text{WfType}∧ \\
\end{array}
\]

\[
\Sigma \vdash c : C \text{ contype} \quad \text{Under (prefix) context } \Sigma, \text{ the typing } c : C \text{ refines a typing in } \mathcal{U}
\]

\[
\Sigma \vdash A \text{ type} \\
\Sigma \vdash s \text{ type} \\
(c : \tau \to d) \in \mathcal{U} \\
\Sigma \vdash (A \to s) \sqsubset (\tau \to d) \quad \text{ContypeArr}
\]

\[s \in S \quad \text{WfTypeSort}
\]

\[\Sigma, S(K), \Sigma \vdash s \text{ type}
\]

\[\Sigma \vdash c : A \to s \text{ contype}
\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5.png}
\caption{Type well-formedness}
\end{figure}

\[
\Sigma \vdash s \preceq t \text{ and subtyping } \Sigma \vdash A \preceq B, \text{ are independent of variable typing and don’t include } \Gamma \text{ at all.}
\]

Traditional formulations of refinements assume the signature is given once at the beginning of the program. Since the same signature is used throughout a given typing derivation, the signature can be omitted from the typing judgments. In this paper, our goal is to support extensible refinements, where the signature can evolve within a typing derivation; in this respect, the signature is analogous to an ordinary typing context \(\Gamma\), which is extended in subderivations that type \(\lambda\)-expressions and other binding forms. So the signature must be explicit in our judgment forms.

Constructor types \(C\) are types of the form \(A \to s\). In past formulations of datasorts, constructor types in the signature use intersection to represent multiple behaviours. For example, a “one” constructor for bitstrings, which represents appending a 1 bit, takes odd-parity bitstrings to even-parity and vice versa; its type in the signature is the intersection type \((\text{odd} \to \text{even}) \land (\text{even} \to \text{odd})\). Such a formulation ensures that the signature has a standard property of (typing) contexts: each data constructor is declared only once; additional behaviours are conjoined (intersected) within a single declaration \(c : C_1 \land C_2 \land \cdots\). In our setting, we must be careful about not only which types a constructor has, but when those types were declared. The reasons are explained below; for now, just note that we will write something like \(c : C_1, \ldots, c : C_2\) rather than \(c : C_1 \land C_2\).

\textit{Structure of signatures.} A signature \(\Sigma\) is a sequence of \textit{blocks} \(S(K)\) of declarations, where refinements declared in outer scopes in the program appear to the left of those declared in inner scopes.

Writing \((s \sqsubseteq d)/(K)\) declares \(s\) to be a sort refining some (unrefined) datatype \(d\); however, we usually elide the datatype and write just \(s(K)\). The declarations \(K\), called the \textit{block} of \(s\), define the values (constructors) of \(s\), and the subsortings for \(s\). Declarations outside this block may declare new subsorts and supersorts of \(s\) only if doing so would not affect \(s\)—for example, adding inhabitants to
s via a constructor declaration, or declaring a new sub-sorting between s and
previously declared sorts, would affect s and will be forbidden (via signature
well-formedness). The grammar generalizes this construct to multiple sorts, e.g.
(s_1 ⊆ d_1, s_2 ⊆ d_2)\langle K \rangle, abbreviated as \langle s_1, s_2 \rangle\langle K \rangle.

Writing \langle s_1 \preceq s_2 \rangle says that s_1 is a sub-sort of s_2, and \langle c : C \rangle says that constructor
c has type C, where C has the form A \rightarrow s. A constructor c can be given more
than one type: \Sigma = \langle s, s_1, s_2 \rangle \langle s_1 \preceq s, s_2 \preceq s, c : s_1 \rightarrow s_2, c : s_2 \rightarrow s_1 \rangle.

Adding inhabitants to a sort is only allowed within its block. Thus, the
following signature is ill-formed, because c' : 1 \rightarrow s adds the value c'() to s,
but c' : 1 \rightarrow s is not within s's block: \langle c : s \rightarrow s \rangle, t \langle c' : 1 \rightarrow s \rangle. New sorts can be
declared as subsorts and supersorts of each other, and of previously declared
sorts: \langle s_1 \preceq s, c_2 : 1 \rightarrow s \rangle, t \langle t \preceq s, c_2 : 1 \rightarrow t \rangle.

However, a block cannot modify the subsorting relation between earlier sorts;
“backpatching” \langle s_1 \preceq s_2 \rangle into the first block, through a new intermediate sort t, is
not permitted: The signature \Sigma_\ast = \langle s_1, s_2 \rangle \langle c : 1 \rightarrow s_1, c : 1 \rightarrow s_2 \rangle, t \langle s_1 \preceq t, t \preceq s_2 \rangle
is not permitted even though it looks safe: sorts s_1 and s_2 have the same set of
inhabitants—the singleton set \{c()\}—so the values of s_1 are a subset of the values
of s_2. But this fact was not declared in the first block, which is the definition
of s_1 and s_2. We assume the declaration of the first block completely reflects
the programmer’s intent: if they had wanted structural subsorting, rather than
nominal subsorting, they should have declared \langle s_1 \preceq s_2 \rangle in the first block. Allowing
backpatching would not violate soundness, but would reduce the power of the
type system: nominal subsorting would no longer be supported, since it could
be made structural after the fact.

Ordering. A block S(K) can refer to the sorts S being defined and to sorts
declared to the left. In contrast to block ordering, the order of declarations
inside a block doesn’t matter.

### 3.4 Introduction form

From a type-theoretic perspective, the first questions about a type are: (1) How
are the type’s inhabitants created? That is, what are the type’s introduction
rules? (2) How are its inhabitants used? That is, what are its elimination rules?
(Gentzen (1934) would ask the questions in this order; the reverse order has
been considered by Dummett, among others (Zeilberger 2009).) In our setting,
we must also ask: What happens with the introduction and elimination forms
when new refinements are introduced?

In the introduction rule—\textbf{DataI} in Figure 6—the signature \Sigma is separated from
the ordinary context \Gamma (which contains typing assumptions of the form x : A). The
typing of c is delegated to its first premise, \Sigma \vdash c : A \rightarrow s, so we need a way
to derive this judgment. At the top of Figure 6, we define a single rule \textbf{ConArr},
which looks up the constructor in the signature and weakens the result type
codomain), expressing a subsumption principle. (Since we’ll have subsumption
as a typing rule, including it here is an unforced choice; its presence is meant to
make the metatheory of constructor typing go more smoothly.)
In a system of extensible refinements, adding refinements to a signature should preserve typing. That is, if \( \Sigma; \Gamma \vdash e : B \), then \( \Sigma, \Sigma'; \Gamma \vdash e : B \). This is a weakening property: we can derive, from the judgment that \( e \) has type \( B \) under \( \Sigma \), the logically weaker judgment that \( e \) has type \( B \) under more assumptions \( \Sigma, \Sigma' \). (The signature becomes longer, therefore stronger; but a turnstile is a kind of implication with the signature as antecedent, so the judgment becomes weaker, hence “weakening”.) So for the introduction form, we need that if \( \Sigma \vdash c : A \rightarrow s \), then \( \Sigma, \Sigma' \vdash c : A \rightarrow s \). Under our formulation of the signature, this holds: If \( c : A \rightarrow s \), then there exists \( (c : A \rightarrow s') \in \Sigma \) such that \( s' \leq s \). Therefore, there exists \( (c : A \rightarrow s') \in (\Sigma, \Sigma') \). Likewise, since \( \Sigma \vdash s' \leq s \), we also have \( \Sigma, \Sigma' \vdash s' \leq s \). One cannot use \( \Sigma' \) to withdraw a commitment made in \( \Sigma \).

### 3.5 Elimination form: case expressions

Exhaustiveness checking for case expressions assumes complete knowledge about the inhabitants of types. Thus, we must avoid extending a signature in a way that adds inhabitants to previously declared sorts. Consider the case expression \( \text{case } x : \text{empty of } \text{Nil}() \Rightarrow () \) which is exhaustive for the signature \( \Sigma = (\text{list}, \text{empty})(\text{empty} \leq \text{list}, \text{Nil} : 1 \rightarrow \text{empty}, \text{Cons} : \text{list} \rightarrow \text{list}) \) but not for

\[
(\Sigma, \Sigma') = (\text{list}, \text{empty})(\text{empty} \leq \text{list}, \text{Nil} : 1 \rightarrow \text{empty}, \text{Cons} : \text{list} \rightarrow \text{list}),
\]

\[
(\text{Cons} : \text{list} \rightarrow \text{empty})
\]

Suppose we type-check the case expression under \( \Sigma \), but then extend \( \Sigma \) to \( (\Sigma, \Sigma') \). Evaluating the above case expression with \( x = \text{Cons}(\text{Nil}()) \) will “fall off the end”. The inversion principle that “every \text{empty} has the form \text{Nil}()” is valid under \( \Sigma \), but with the additional type for \( \text{Cons} \) in \( \Sigma' \), that inversion principle becomes invalid under \( (\Sigma, \Sigma') \). Our system will reject the latter signature as ill-formed.

In the following, “up” and “down” are used in the usual sense: a subsort is below its supersort. In \( \Sigma' \), the second constructor type for \( \text{Cons} \) had a smaller codomain than the first: the second type had \text{empty}, instead of \text{list}. Varying the codomain downward can be sound when the lower codomain is newly defined: \( \Sigma, \Sigma'' = \Sigma, \text{subempty} (\text{subempty} \leq \text{empty}, \text{Nil} : 1 \rightarrow \text{subempty}) \). Here, the inversion principle that every \text{empty} is \text{Nil} is still valid (along with the new inversion principle that every \text{subempty} is \text{Nil}). We only added information about a new sort \text{subempty}, without changing the definition of \text{list} and \text{empty}.

**Moving the domain down.** Giving a new type whose domain is smaller, but that has the same codomain, is sound but pointless. For example, extending \( \Sigma \) with \( \text{Cons} : \text{empty} \rightarrow \text{list} \), which is the same as the type \( \Sigma \) has for \( \text{Cons} \) except that the
domain is empty instead of list, is sound. The inversion principle for values v of type list in Σ alone is “either (1) v has the form Nil() or (2) v has the form Cons(y) where y has type list”. Reading off the new inversion principle for list from Σ, Cons : empty→list, we get “either (1) v has the form Nil(), or (2) v has the form Cons(y) where y has type list, or (3) v has the form Cons(y) where y has type empty”. Since empty is a subset of list, part (3) implies part (2), and any case arm that checks under the assumption that y : list must also check under the assumption that y : empty. Here, the new signature is equivalent to Σ alone; the “new” type for Cons is spurious.

Moving the codomain up. Symmetrically, giving a new type whose codomain gets larger is sound but pointless. For example, adding Nil : 1→list to Σ has no effect, because (in the introduction form) we could use the old type Nil : 1→empty with subsumption (empty ≤ list).

Moving the domain up. Making the domain of a constructor larger is unsound in general. To show this, we need a different starting signature Σ₂.

\[ Σ₂ = (\text{list, empty, nonempty})(\text{empty} ≤ \text{list, nonempty} ≤ \text{list, Nil : 1→empty, Cons : empty→nonempty}) \]

This isn’t a very useful signature—it doesn’t allow construction of any list with more than one element—but it is illustrative. We can read off from Σ₂ the following inversion principle for values v of sort nonempty: “v has the form Cons(y) where y has type empty”. If x : nonempty then case x of Cons(Nil(λ)) ⇒ (2) is exhaustive under Σ₂. Now, extend Σ₂: Σ₂, Σ₂ ′ = Σ₂, (Cons : list→nonempty). For the signature Σ₂, Σ₂ ′, the inversion principle for nonempty should be “(1) v has the form Cons(y) where y has type empty, or (2) v has the form Cons(y) where y has type list”. But there are more values of type list than of type empty. The new inversion principle gives less precise information about the argument y, meaning that the old inversion principle gives more precise information than (Σ₂, Σ₂ ′) allows. Concretely, the case expression above was exhaustive under Σ₂, but is not exhaustive under (Σ₂, Σ₂ ′) because Cons(Cons(Nil(λ))) has type list.

The above examples show that signature extension can be sound but useless, unsound, or sound and useful (when the domain and codomain, or just the codomain, are moved down). Ruling out unsoundness will be the main purpose of our type system, where unsoundness includes raising a “match” exception due to a nonexhaustive case. The critical requirement is that each block must not affect previously declared sorts by adding constructors to them, or by adding subsortings between them.

3.6 Typing

Figure 3 gives rules deriving the main typing judgment Σ;Γ ⊢ e : A. The variable rule Var, the introduction (→I) and elimination (→E) rules for →, and the introduction rules for the unit type (1I) and products (11) are standard. Products can be eliminated via case e of (x₁, x₂) ⇒ · · ·, so they need no elimination rule.
Subsumption. A subsumption rule $\text{Sub}$ incorporates subtyping, based on the subsort relation $\preceq$; see Section 3.7. Several of the subtyping rules express the same properties as elimination rules would; for example, anything of type $A_1 \land A_2$ has type $A_1$ and also type $A_2$. Consequently, we can omit these elimination rules without losing expressive power.

Intersection. The introduction rule $\text{Arr}$ corresponds to a binary version of the introduction rule for parametric polymorphism in System F. The restriction to a value $v$ avoids unsoundness in the presence of mutable references (Davies and Pfenning 2000), similar to SML’s value restriction for parametric polymorphism (Wright 1995). We omit the elimination rules, which are admissible using $\text{Sub}$ and subtyping (Section 3.7).

Datasorts. Rule $\text{DataE}$ introduces a datasort, according to a constructor type found in $\Sigma$ (via the $\Sigma \vdash c : C$ judgment). Rule $\text{DataE}$ examines an expression $e$ of type $A$ and checks matches $ms$ under the assumption that the expression matches the wildcard pattern $\_$. see Section 3.8.

Re-refinement. Rule $\text{Declare}$ allows sorts to be declared. Its premises check that (1) the signature $\Sigma'$ is a valid extension of $\Sigma$ (see Section 3.8); (2) the type $B$ of the expression is well-formed without the extension $\Sigma'$, which prevents sorts declared in $\Sigma'$ from escaping their scope; (3) that the expression $e$ is well-typed under the extended signature $(\Sigma, \Sigma')$.  

![Typing rules for constructors and expressions](Fig. 6.)
3.7 Subtyping

Our subtyping judgment $\Sigma \vdash A \leq B$ says that all values of type $A$ also have type $B$. The rules follow the style of [Dunfield and Pfenning, 2003]; in particular, the rules are orthogonal (each rule mentions only one kind of connective) and transitivity is admissible. Instead of an explicit transitivity rule, we bake transitivity into each rule; for example, rule $\text{≤ L1}$ has a premise $A_1 \leq B$ and conclusion $(A_1 \land A_2) \leq B$, rather than just $(A_1 \land A_2) \leq A_1$ (with no premises). This makes the rules easier to implement: to decide whether $A \leq C$, we never have to guess a middle type $B$ such that $A \leq B$ and $B \leq C$.

3.8 Signature well-formedness

A signature is well-formed if standard conditions (e.g. no duplicate declarations of sorts) and conservation conditions hold. Reading Figure 7 from bottom to top, we start with well-formedness of signatures $\Sigma$. For each block $\text{SigBlock}$, rule $\text{SigBlock}$ checks that the sorts $S$ are not duplicates ($S \cap \text{dom}(\Sigma) = \emptyset$), and then checks that (1) subsorting is conserved by $K$ and (2) each element in $K$ is safe.

(1) Subsorting preservation. The subsortings declared in $K$ must not affect the subsort relation between sorts previously declared in $\Sigma$. The left-to-right direction of this “iff” always holds by weakening: adding to a signature cannot delete edges in the subsort relation. The right-to-left direction is contingent on the contents of $K$; see signature $\Sigma_1$ in Section 3.3. This premise could also be written as $(\Sigma \vdash \preceq_{\text{dom}(\Sigma)}) = (\Sigma, S(K) \vdash \preceq_{\text{dom}(\Sigma)})$, where $\preceq_{\text{dom}(\Sigma)}$ is the $\preceq$ relation restricted to sorts in $\text{dom}(\Sigma)$.

(2a) Subsort elements. Rule $\text{BlockSubsort}$ checks that the subsorts are in scope.

(2b) Constructor element safety. Rule $\text{BlockCon}$ is first premise checks that $s \in S$. (Certain declarations with $s \notin S$ would be safe, but useless.) Its second premise checks that the constructor type $A \rightarrow s$ is well-formed. Finally, for all sorts $t$ that were (1) previously declared (in $\text{dom}(\Sigma)$) and (2) supersorts of the constructor’s codomain ($s \preceq t$), the rule checks that the constructor is “safe at $t$”. The judgment $\Sigma; S(K) \vdash c : A \rightarrow s$ safe at $t$ says that adding the constructor typing $c : A \rightarrow s$ does not invalidate $\Sigma$’s inversion principle for $t$. Rule $\text{SafeConAt}$ checks that signature $\Sigma$ already has a constructor typing $c : A' \rightarrow s'$, where $s' \preceq t$, such that $A \leq A'$. Thus, any value $c(v)$ typed using $c : A \rightarrow s$ can already be
Domain (declared sorts) of \( \Sigma \):
\[
\text{dom}(\Sigma) = \text{dom}(S_1(\langle K_1 \rangle), \ldots, S_n(\langle K_n \rangle)) = S_1 \cup \cdots \cup S_n
\]

Sort \( s_1 \) is a subset of \( s_2 \)
\[
\Sigma \vdash s_1 \subseteq s_2
\]

\[
\begin{align*}
\Sigma \vdash s \subseteq s & \equiv \text{Ref} \\
\Sigma \vdash s_1 \subseteq s_2 & \equiv \text{Assum} \\
\Sigma \vdash s_1 \subseteq s_2 & \equiv \text{Trans}
\end{align*}
\]

\( \Sigma ; S(\langle K \rangle) \vdash c : C \text{ safe at } t \) C safely extends a type given by \( \Sigma \) for \( c \)
\[
\begin{align*}
\Sigma, S(\langle K \rangle) \vdash c : A' \text{ safe at } t & \equiv \text{SafeConAt} \\
\Sigma ; S(\langle K \rangle) \vdash K_{\text{elem}} \text{ safe} & \equiv \text{KelemSafe}
\end{align*}
\]

\( \Sigma ; S(\langle K \rangle) \vdash K_{\text{elem}} \in K \text{ is safe for } \Sigma ; S(\langle K \rangle) \)
\[
\begin{align*}
\Sigma, S(\langle K \rangle) \vdash c : A \rightarrow s \text{ safe at } t & \equiv \text{SafeContype} \\
\end{align*}
\]

\( s_1, s_2 \in (\text{dom}(\Sigma) \cup S) \)
\[
\Sigma ; S(\langle K \rangle) \vdash s_1 \subseteq s_2 \text{ safe} & \equiv \text{BlockSubsort}
\]

\( \Sigma \text{ sig} \)
\[
\Sigma \text{ sig for all } t_1, t_2 \in \text{dom}(\Sigma),
\begin{align*}
(S \cap \text{dom}(\Sigma)) & = \emptyset & \text{SigEmpty} \\
(\Sigma \vdash t_1 \subseteq t_2) & \text{SigBlock}
\end{align*}
\]

Fig. 8. Signature well-formedness and subsorting

\( \Sigma \) is well-formed and \( \Sigma \) is safe for all \( t \), so the new constructor typing \( c : A \rightarrow s \) does not add inhabitants to \( t \).

This check is not analogous to function subtyping, because we need covariance (\( A \leq A' \)), not contravariance. The relation \( \subseteq \) (Figure 3) is a closer analogy.

More subtly, \( \text{SafeContype} \) also checks that \( t \) is a subset of \( s \). Suppose we have the signature \( \Sigma = \{ t, s_1, s_2 \} ; (s_1 \leq t, s_2 \leq t, c_1 : s_1, c_2 : s_2) \) and extend it with \( s'(s \leq t, c_1 : s) \).

To focus on the issue at hand, we assume \( c_1 \) and \( c_2 \) take no arguments.) For the original signature \( \Sigma \), the inversion principle for \( t \) is: If a value \( v \) has type \( t \), then either \( v = c_1 \) and \( v \) has type \( s_1 \), or \( v = c_2 \) and \( v \) has type \( s_2 \). However, under the extended signature, there is a new possibility: \( v \) has type \( s \). Merely being inhabited by \( c_1 \) is not sufficient to allow \( s \) to be a subset of \( t \).

If, instead, we start with \( \Sigma' = \{ t, s_1, s_2 \} ; (c_1 : t, s_1 \leq t, s_2 \leq t, c_1 : s_1, c_2 : s_2) \) then the inversion principle for \( t \) under \( \Sigma' \) is that \( v \) has type \( s_1 \), type \( s_2 \), or type \( t \).

Therefore, any case arm whose pattern is \( x \) as \( c_1 \) must be checked assuming \( x : t \). If an expression can be typed assuming \( x : t' \), then it can be typed assuming \( x : t' \) for any \( t' \leq t \), so the inversion principle (again, under \( \Sigma' \) before extension) is equivalent to “\( v \) has type \( t' \).” Extending \( \Sigma' \) with \( s(s \leq t, c_1 : s) \) would extend the inversion principle to say “if \( v : t \) then \( v \) has type \( t \), or \( v \) has type \( s' \), but since \( s \leq t \) the extended inversion principle is equivalent to that for \( t \) under \( \Sigma' \).
The $s' \preceq s'$ premise of SafeConAt is needed to prove the constructor lemma (Lemma 12), which says that a constructor typing in an extended signature must be below a constructor typing in the original signature.

4 Typing Pattern Matching

Pattern matching is how a program gives different answers on different inputs. A key motivation for datasort refinements is to exclude impossible patterns, so that programmers can avoid having to choose between writing impossible case arms (that raise an “impossible” exception) and ignoring nonexhaustiveness warnings. The pattern typing rules must model the relationship between datasorts and the operational semantics of pattern matching. It’s no surprise, then, that in datasort refinement systems, case expressions lead to the most interesting typing rules.

The relationship between types and patterns is more involved than with, say, Damas–Milner plus inductive datatypes: with (unrefined) inductive datatypes, all the information needed to check for exhaustiveness (also called coverage) is immediately available as soon as the type of the scrutinee is known. Moreover, types for pattern variables can be “read off” by traversing the pattern top-down, tracking the definition of the scrutinee’s inductive datatype. But with datasorts, a set of patterns that looks nonexhaustive at first glance—looking only at the head constructors—may in fact be exhaustive, thanks to the inner patterns.

Giving types to pattern variables is also tricky, because sufficiently precise types may be evident only after examining the whole pattern. For example, when matching $x : \text{bits}$ against the pattern $y \text{ as One(Empty)}$, we shouldn’t settle on $y : \text{bits}$ because the scrutinee $x$ has type bits; we should descend into the pattern and observe that $\text{Empty} : \text{even}$ and $\text{One} : (\text{even} \rightarrow \text{odd})$, so $y$ must have type odd.

Restricting the form of case expressions to a single layer of clearly disjoint patterns $c_1(x_1) | \ldots | c_n(x_n)$ would simplify the rules, at the cost of a big gap between theory and practice: Since real implementations need to support nested patterns, the theory fails to model the real complexities of exhaustiveness checking and pattern variable typing. Giving code examples becomes fraught; either we flatten case expressions (resulting in code explosion), or we handwave a lot.

Another option is to support the full syntax of case expressions, except for as-patterns, so that pattern variables occur only at the leaves. If sub-sorting were always structural, as in Davies’s system, we could exploit a handy equivalence between patterns and values: if the pattern is $x \text{ as c(p}_0\text{)}$, let-bind $x$ to $c(p_0)$ inside the case arm, letting rule DataI figure out the type of $x$. But with nominal sub-sorting, constructing a value is not equivalent; see Davies (2005, pp. 234–5) and Dunfield (2007b, pp. 112–3).

Our approach is to support the full syntax, including as-patterns. This approach was taken by Dunfield (2007b, Chapter 4), but our system seems simpler—partly because (except for signature extension) our type system omits indexed types and union types, but also because we avoid embedding typing derivations inside derivations of pattern typing.
Pattern $p$ is suitable for values of unrefined type $\tau$

\[
\Gamma \vdash p : \tau \quad \text{p-Wild} \quad \Gamma \vdash (x \mathit{as} \ p) : \tau \quad \text{p-As} \quad \Gamma \vdash \emptyset : \tau \quad \text{p-Empty} \quad \Gamma \vdash () : 1 \quad \text{p-Unit}
\]

\[
\Gamma \vdash p_1 : \tau \quad \text{p-Or} \quad \Gamma \vdash p_2 : \tau 
\]

\[
\Gamma \vdash (p_1 \cup p_2) : \tau \quad \text{p-Pair} \quad \Gamma \vdash c : (\tau \rightarrow d) \quad \text{p-Con}
\]

Figure 9. Pattern type rules

\[
\Sigma; \Gamma; p : A \vdash ms : D \quad \text{For a scrutinee of type } A \text{ that matches residual pattern } p,
\]

check each match in $ms$ against $D$

\[
\forall (\Gamma' \vdash B) \quad \Sigma \vdash A \subseteq \tau \quad \in \text{intersect}(\Sigma \vdash A; p \cap p_1) : \Sigma, \Gamma; \Gamma' \vdash e_1 : D 
\]

\[
\Sigma; \Gamma; p : A \vdash ((p_1 \Rightarrow e_1) \mid ms) : D \quad \text{TypeMs}
\]

\[
\text{intersect}(\Sigma \vdash A; p) = \emptyset \quad \Sigma, \Gamma; p : A \vdash \emptyset : D \quad \text{TypeMsEmpty}
\]

Figure 10. Match typing

Instead, we confine most of the complexity to a single mechanism: a function called $\text{intersect}$, which returns a set of types (and contexts that type as-variables) that represent the intersection between a type and a pattern. The definition of this function is not trivial, but does not refer to expression-level typing.

4.1 Unrefined pattern typing, match typing, and pattern operations

Figure 9 defines a judgment $\Gamma \vdash p : \tau$ that says that pattern $p$ matches values of unrefined type $\tau$ under the unrefined signature $\Gamma$.

Rule $\text{DataE}$ for case expressions (Figure 6) invokes a match typing judgment, $\Sigma; \Gamma; p : A \vdash ms : D$. In this judgment, $p$ is a residual pattern that represents the space of possible values. For the first arm in a case expression, no patterns have yet failed to match, so the residual pattern in the premise of $\text{DataE}$ is $\emptyset$.

Each arm, of the form $p_1 \Rightarrow e_1$, is checked by rule $\text{TypeMs}$ (Figure 10). The leftmost premises check that the type $A$ corresponds to the pattern type $\tau$. The middle “for all” checks $e_1$ under various assumptions produced by the $\text{intersect}$ function (Section 4.2) with respect to the pattern $p \cap p_1$, ensuring that if $p_1$ matches the value at run time, the arm is well-typed. The last premise moves on to the remaining matches; there, we know that the value did not match $p_1$, so we subtract $p_1$ from the previous residual pattern $p$—expressed as $p \cap \neg p_1$.

These operations are defined in the appendix (Figure 13).

When typing reaches the end of the matches, $ms = \emptyset$ in rule $\text{TypeMsEmpty}$, we check that the case expression is exhaustive by checking that $\text{intersect}$ returns $\emptyset$. For case expressions that are syntactically exhaustive, such as a case expression
Intersection of type $A$ with pattern $p$ where each $B^*$ has the form $(\Gamma^\prime \vdash B^\prime)$:

$$\text{intersect}(\Sigma \vdash A; p) = B^*$$

intersections in $p$:

$$\text{intersect}(\Sigma \vdash A; \bot) = \{(\cdot \vdash A)\}$$

$$\text{intersect}(\Sigma \vdash A; \emptyset) = \emptyset$$

$$\text{intersect}(\Sigma \vdash A; x \text{ as } p) = \{(\Gamma', x : B \vdash B) \mid (\Gamma^\prime \vdash B) \in \text{intersect}(\Sigma \vdash A; p)\}$$

$$\text{intersect}(\Sigma \vdash A_1 \land A_2; (p_1, p_2)) = \{(\Gamma_1, \Gamma_2 \vdash B_1 \land B_2) \mid (\Gamma_1 \vdash B_1) \in \text{intersect}(\Sigma \vdash A_1; p_1) \land (\Gamma_2 \vdash B_2) \in \text{intersect}(\Sigma \vdash A_2; p_2)\}$$

$$\text{intersect}(\Sigma \vdash s; c(p_0)) = \{(\Gamma^\prime \vdash s_c) \mid (c : A_c \to s_c) \in \Sigma \land \Sigma \vdash s_c \leq s \land (\Gamma^\prime \vdash B) \in \text{intersect}(\Sigma \vdash A_c; p_0)\}$$

Fig. 11. Intersection of a type with a pattern

over lists that has both Nil and Cons arms, the residual pattern $p$ will be the empty pattern $\emptyset$; the intersect function on an empty pattern returns $\emptyset$.

We define pattern complement $\bar{c}$ and pattern intersection $c_1 \cap c_2$ in the appendix (Figure 13). For example, $\bar{\bot} = \emptyset$. No types appear in these definitions, but the complement of a constructor pattern $c(p_0)$ uses the (implicit) unredefined signature $\mathcal{U}$. Our definition of pattern complement never generates as-patterns, so we need not define intersection for as-patterns.

4.2 The intersect function

We define a function intersect that builds the “intersection” of a type and a pattern. Given a signature $\Sigma$, type $A$ and pattern $p$, the intersect function returns a (possibly empty) set of tracks $((\Gamma_1 \vdash B_1), \ldots, (\Gamma_n \vdash B_n))$. Each track $(\Gamma^\prime \vdash B)$ has a list of typings $\Gamma^\prime$ (giving the types of $\text{as}$-variables) and a type $B$ that represents the subset of values inhabiting $A$ that also match $p$. The union of $B_1$ through $B_n$ constitutes the intersection of $A$ and $p$. We call these “tracks” because each one represents a possible shape of the values that match $p$, and the type-checking “train” must check a given case arm under each track’s $\Gamma^\prime$.

Many of the clauses in the definition of intersect (see Figure 10) are straightforward. The intersection of $A$ with the wildcard $\bot$ is just $\{(\cdot \vdash A)\}$. Dually, the intersection of $A$ with the empty pattern $\emptyset$ is the empty set. In the same vein, the intersection of $A$ with the or-pattern $p_1 \lor p_2$ is the union of two intersections ($A$ with $p_1$, and $A$ with $p_2$). The intersection of a product $A_1 \land A_2$ with a pair pattern is the union of products of the pointwise intersections.

The most interesting case is when we intersect a sort $s$ with a pattern of the form $c(p_0)$. For this case, intersect iterates through all the constructor declarations in $\Sigma$ that could have been used to create the given value: those of the form $(c : A_c \to s_c)$ where $s_c \leq s$. For each such declaration, it calls intersect on $A_c$ and $p_0$. For each resulting track $(\Gamma^\prime \vdash B)$, it returns a track $(\Gamma^\prime \vdash s_c)$.

Optimization. In practice, it may be necessary to optimize the result of intersect. If $\Sigma = \{\text{list, empty} \land \text{empty} \leq \text{list}, \text{Nil : 1} \to \text{empty}, \text{Cons : empty} \to \text{list}, \text{Cons : list} \to \text{list}\}$ then intersect$(\Sigma \vdash \text{Cons}(x \text{ as } \bot); \text{list})$ returns $\{(x : \text{empty} \vdash \text{list}), (x : \text{list} \vdash \text{list})\}$. 
Since any case arm that checks under \( x : \text{list} \) will check under \( x : \text{empty} \), there is no point in trying to check under \( x : \text{empty} \). Instead, we should check only under \( x : \text{list} \). A similar optimization in the Stardust type checker could reduce the size of the set of tracks by “about an order of magnitude” (Dunfield 2007b, p. 112).

**Missing clauses?** As is standard in typed languages, pattern matching doesn’t look inside \( \lambda \), so \text{intersect} needs no clause for \( \to/\lambda \). If we can’t match on an arrow type, we don’t need to match on intersections of arrows. The other useful case of intersection is on sorts, \( s_1 \land s_2 \). However, an intersection of sorts can be obtained by declaring a new sort below \( s_1 \) and \( s_2 \) with the appropriate constructor typings, so we omit such a clause from the definition.

**Comparison to an earlier system.** A declarative system of rules in Dunfield (2007b, Chapter 4) appears to be a conservative extension of \text{intersect}: the earlier system supports a richer type system, but for the features in common, the information produced is similar to that of \text{intersect}. The earlier system was based on a judgment \( \Sigma \vdash p \leftarrow A (e \leftarrow D) \). To clarify the connection to the present system, we adjust notation; for example, we make \( \Sigma \) explicit.

The meta-variables \( \Sigma \), \( p \), and \( A \) directly correspond to the arguments to \text{intersect}, while \( e \) and \( D \) correspond to \( e_1 \) and \( D \) in our rule [TypeMs]. No meta-variables correspond directly to the tracks in the result of \text{intersect}, but within \( \Sigma \vdash p \leftarrow A (e \leftarrow B) \), we find subderivations of \( B + \Gamma \vdash \text{forgettype} \leftarrow e \leftarrow D \), where the set of pairs \( (\Gamma, B) \) indeed correspond to the result of \text{intersect}.

Cutting through the differences in the formalism, and omitting rules for unions and other features not present in this paper, the earlier system behaves like \text{intersect}. For example, \((p_1, p_2)\) was also handled by considering each component, and assembling all resulting combinations. Perhaps most importantly, \( c(p_0) \) was also handled by considering each constructor type in the signature, filtering out inappropriate codomains, and recursing on \( p_0 \). A rule for \( \land \) appears in the declarative system in Dunfield (2007b, Chapter 4), but the rule was never implemented, and seems not to be needed in practice.

Since the information given by the older system is precise enough to check interesting invariants of actual programs, our definition of \text{intersect} should also be precise enough.

## 5 Operational Semantics

We prove our results with respect to a call-by-value, small-step operational semantics. The main judgment form is \( e \rightarrow e' \), which uses evaluation contexts \( E \). Stepping case expressions is modelled using a judgment \( ms \rightarrow_{\nu} e' \), which compares each pattern in \( ms \) against the value \( \nu \) being cased upon. This comparison is handled by the judgment \( p \text{ match} \nu \rightarrow \theta \), which says that \( \theta \) is evidence that \( p \) matches \( \nu \) (that is, \( [\theta]p = \nu \)). The rules are in Figure 14 in the appendix.
6 Metatheory

This section gives definitions, states some lemmas and theorems, and discusses their significance in proving our main results. For space reasons, we summarize a number of lemmas; their full statements appear in the appendix. All proofs are also relegated to the appendix.

Subtyping and sorting. Subtyping is reflexive and transitive (Lemmas 6, 7).

We define what it means for signature extension to preserve subtyping:

Definition 1 (Preserving subtyping). Given \( \Sigma_1 \) and \( \Sigma_2 \), we say that \( \Sigma_2 \) preserves subtyping of \( \Sigma_1 \) iff for all sorts \( s, t \in \text{dom}(\Sigma_1) \), if \( \Sigma_1, \Sigma_2 \vdash s \preceq t \) then \( \Sigma_1 \vdash s \preceq t \).

This definition allows new sorts in \( \text{dom}(\Sigma_2) \) to be subsorts or supersorts of the old sorts in \( \text{dom}(\Sigma_1) \), provided that the subsort relation between the old sorts doesn’t change.

If two signatures do not have subsortings that cross into each other’s domain, they are non-adjacent; non-adjacent signatures preserve subtyping.

Definition 2 (Non-adjacency). Two signatures \( \Sigma_1 \) and \( \Sigma_2 \) are non-adjacent iff each signature contains no subsortings of the form \( s_1 \preceq s_2 \) or \( s_2 \preceq s_1 \), where \( s_1 \in \text{dom}(\Sigma_1) \) and \( s_2 \in \text{dom}(\Sigma_2) \).

Theorem 1 (Non-adjacent preservation). If \( \Sigma_2 \) preserves subtyping of \( \Sigma_1 \) and \( \Sigma_3 \) preserves subtyping of \( \Sigma_1 \) and \( \Sigma_2 \) and \( \Sigma_3 \) are non-adjacent then \( \Sigma_3 \) preserves subtyping of \( (\Sigma_1, \Sigma_2) \).

Strengthening, weakening, and substitution. Theorem 4 (Weakening) will allow the assumptions in a judgment to be changed in two ways: (1) the signature may be strengthened by replacing a signature \( (\Sigma, \Sigma') \) with a signature \( (\Sigma, \Omega, \Sigma') \); and (2) the context may be strengthened by replacing \( \Gamma \) with a context \( \Gamma' \) in which any typing assumption \( (x : A) \in \Gamma \) can be replaced with \( (x : A^+) \in \Gamma' \), if \( A \leq A^+ \). Repeatedly applying (1) with different \( \Omega \) leads to a more general notion of strengthening a signature:

Definition 3. A signature \( \Sigma' \) is stronger than \( \Sigma \), written \( \Sigma' \leq_{\text{sig}} \Sigma \), if \( \Sigma' \) can be obtained from \( \Sigma \) by inserting entire signatures at any position in \( \Sigma \).

We often use the less general notion (inserting a single \( \Omega \)), which simplifies proofs. For any result stated less generally, however, the more general strengthening of Definition 3 can be shown by induction on the number of blocks inserted.

Definition 4. Under \( \Sigma \), a context \( \Gamma' \) is stronger than \( \Gamma \), written \( \Gamma \vdash \Gamma' \leq_{\text{ctx}} \Gamma \), if for each \( (x : A') \in \Gamma' \), there exists \( (x : A) \in \Gamma \) such that \( \Sigma \vdash A' \leq A \).

Several lemmas show weakening. Lemma 8 says that \( \Sigma \) in \( \Sigma \vdash J \) can be replaced by a stronger \( \Sigma' \), where \( J \) has the form \( A \text{ type} \) or \( s_1 \leq s_2 \) or \( A \preceq B \) or \( c : A \to s \) or \( A \sqcup \tau \) or \( c : C \). Lemma 9 says that \( (\Sigma, \Omega, \Sigma') \) can replace \( (\Sigma, \Sigma') \) in \( \Sigma, \Sigma'; S(K) \vdash c : A \to s \) safe at \( t \). Lemma 10 allows the sort \( t' \) in the judgment \( \Sigma; S(K) \vdash c : A \to s \) safe at \( t' \) to be replaced by a supersort \( t \).

Using the above lemmas and Theorem 4 we can show that the key judgment “\( \cdots c : A \to s \) safe” can be weakened by inserting \( \Omega \) inside the signature:
Theorem 2 (Weakening ‘safe’).
If \((\Sigma, \Sigma') \text{ sig} \text{ and } \Sigma, \Omega \text{ sig and } \text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset \)
and \(\text{dom}(\Sigma, \Omega, \Sigma') \cap S = \emptyset \) and \(K\) does not mention anything in \(\text{dom}(\Omega)\)
and \(S(K)\) preserves subsorting for \((\Sigma, \Sigma')\)
and \((c: A \rightarrow s) \in K\) and \(\Sigma, \Omega, \Sigma' ; S(K) \vdash c: A \rightarrow s\) safe
then \(\Sigma, \Omega, \Sigma'; S(K) \vdash c: A \rightarrow s\) safe.

With this additional lemma, we have weakening for the judgments involved
in checking that a signature is well-formed, so we can show that if \(\Sigma\) is safely
extended by \(\Sigma'\) and separately by \(\Omega\), then \(\Omega\) and \(\Sigma'\), together, safely extend \(\Sigma\).

Theorem 3 (Signature Interleaving).
If \((\Sigma, \Sigma') \text{ sig and } \Sigma, \Omega \text{ sig and } \text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset \) then \((\Sigma, \Omega, \Sigma', \Sigma') \text{ sig}.\)

Ultimately, we will show type preservation; in the preservation case for the
\textbf{Declare} rule, we extend the signature in a premise. We therefore need to show
that the typing judgment can be weakened. Since the typing rules for matches
involve the \textit{intersect} function, we need to show that a stronger input to \textit{intersect}
yields a stronger output; that is, a longer (stronger) signature yields a stronger
type \(B_+\) (a subtype of \(B\)) and a stronger context \(\Gamma_+\) typing as-variables.

\textbf{Definition 5.} Under a signature \(\Sigma\), a track \((\Gamma_+ \vdash B_+)\) is stronger than \((\Gamma \vdash B)\),
written \(\Sigma \vdash (\Gamma_+ \vdash B_+) \leq_{\text{trk}} (\Gamma \vdash B)\), if and only if \(\Sigma \vdash \Gamma_+ \leq_{\text{ctx}} \Gamma\) and \(\Sigma \vdash B_+ \leq B.\)

A set of tracks \(B^+_\Sigma\) is stronger than \(B^+\), written \(B^+_\Sigma \leq_{\text{trk}} B^+\), if and only
if, for each track \((\Gamma_+ \vdash B_+) \in B^+_\Sigma\), there exists a track \((\Gamma \vdash B) \in B^+\) such that
\((\Gamma_+ \vdash B_+) \leq_{\text{trk}} (\Gamma \vdash B)\).

Lemma [13] says that the result of \textit{intersect} on a stronger signature is stronger.

We can then show that weakening holds for the typing judgment itself, along
with substitution typing (defined in the appendix) and match typing.

Theorem 4 (Weakening).
If \((\Sigma, \Sigma') \text{ sig} \text{, } \langle \Sigma, \Omega \rangle \text{ sig, } \text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset \) and \(\Sigma, \Omega, \Sigma', \Gamma \vdash \Gamma^\prime \leq_{\text{ctx}} \Gamma \) then
\begin{enumerate}
\item If \(\Sigma, \Sigma' ; \Gamma \vdash e: A\) then \(\Sigma, \Omega, \Sigma'; \Gamma^\prime \vdash e: A\).
\item If \(\Sigma, \Sigma' ; \Gamma \vdash \theta : \Gamma'\) then \(\Sigma, \Omega, \Sigma', \Gamma^\prime \vdash \theta : \Gamma'\).
\item If \(\Sigma, \Sigma'; \Gamma ; p : A \vdash ms : D\) then \(\Sigma, \Omega, \Sigma'; \Gamma^\prime ; p : A \vdash ms : D\).
\end{enumerate}
\textbf{Properties of values.} Substitution properties (Lemmas [14] and [15]) and inversion
(or \textit{canonical forms}) properties (Lemma [16]) hold.

\textbf{Type preservation and progress.} The last important piece needed for type preservation
is that \textit{intersect} does what it says: if a value \(v\) matches \(p\), then \(v\) has type \(B\)
where \(B\) is one of the outputs of \textit{intersect}.

Theorem 5 (Intersect). If \(\Sigma \text{ sig and } \Sigma; \vdash \nu: A \text{ and } \Sigma \vdash A \text{ type and } p \text{ match } \nu \rightarrow \emptyset \) \text{ and } \text{intersect}(\Sigma; \nu; p) \in B^+ \text{ then there exists } (\Gamma^\prime \vdash B) \in B^+ \text{ s.t. } \Sigma; \vdash \nu: B \) \text{ and } \Sigma; \vdash \theta : \Gamma' \text{ where } \Sigma \vdash B \text{ type and } \Sigma \vdash B \leq A.

The preservation result allows for a longer signature, to model entering the
scope of a \textit{declare} expression or the arms of a \textit{match}. We implicitly assume
that, in the given typing derivation, all types are well-formed under the local signature:
for any subderivation of \(\Sigma; \Gamma \vdash \epsilon' : B\), it is the case that \(\Sigma \vdash B \text{ type}.\)
Theorem 6 (Preservation). If \( \Sigma \) sig \( \vdash \ e : A \) and \( e \mapsto e' \) then there exists \( \Sigma' \) such that \( \Sigma, \Sigma' \vdash e' : A \) where \( (\Sigma, \Sigma') \) sig.

Theorem 7 (Progress). If \( \Sigma \) sig \( \vdash \ e : A \) then \( e \) is a value or there exists \( e' \) such that \( e \mapsto e' \).

7 Bidirectional Typing

The type assignment system in Figure 6 is not syntax-directed, because the rules \( \text{Sub} \) and \( \wedge I \) apply to any shape of expression. Nor is the system directed by the syntax of types: rule \( \text{Sub} \) can conclude \( e : B \) for any type \( B \) that is a supertype of some other type \( A \). Finally, while the choice to apply rule \( \text{DataI} \) is guided by the shape of the expression—it must be a constructor application \( c(e) \)—the resulting sort is not uniquely determined, since the signature can have multiple constructor typings for \( c \).

Fortunately, obtaining an algorithmic system is straightforward, following previous work with datasort refinements and intersection types. We follow the bidirectional typing recipe of Davies and Pfenning (2000); Davies (2005); Dunfield and Pfenning (2004):

1. Split the typing judgment into checking \( \Sigma; \Gamma \vdash e \leftarrow A \) and synthesis \( \Sigma; \Gamma \vdash e \Rightarrow A \) judgments. In the checking judgment, the type \( A \) is input (it might be given via type annotation); in the synthesis judgment, the type \( A \) is output.
2. Allow change of direction: Change the subsumption rule to synthesize a type, then check if it is a subtype of a type being checked against; add an annotation rule that checks \( e \) against \( A \) in the annotated expression \( e : A \).
3. In each introduction rule, e.g. \( \rightarrow I \) make the conclusion a checking judgment; in each elimination rule, e.g. \( \text{DataE} \) make the premise that contains the eliminated connective a synthesis judgment.
4. Make the other judgments in the rules either checking or synthesizing, according to what information is available. For example, the premise of \( \rightarrow I \) becomes a checking judgment, because we know \( B \) from the conclusion.
5. Since the subsumption rule cannot synthesize, add rules such as \( \text{Syn} \wedge \wedge \wedge \text{E} \), which were admissible in the type assignment system.

This yields the rules in Figure 12. (Rules for the match typing judgment \( \Sigma; \Gamma; \pi : A \vdash ms \leftarrow B \) can be obtained from Figure 10 by replacing “:” in “\( e : D \)” and “\( ms : D \)” with “\( \leftarrow \)”.) While this system is much more algorithmic than Figure 6 the presence of intersection types requires backtracking: if we apply a function of type \( (\text{even} \rightarrow \text{odd}) \wedge (\text{odd} \rightarrow \text{even}) \), we need to synthesize \( \text{even} \rightarrow \text{odd} \) first; if we subsequently fail (e.g. if the argument has type \( \text{odd} \)), we backtrack and try \( \text{odd} \rightarrow \text{even} \). Similarly, if the signature contains several typings for a constructor \( c \), we may need to try rule \( \text{ChkDataI} \) with each typing.

Type-checking for this system is almost certainly PSPACE-complete (Reynolds 1996), however, the experience of Davies (2005) shows that a similar system, differing primarily in whether the signature can be extended, is practical if certain techniques, chiefly memoization, are used.
Using these rules, annotations are required exactly on (1) the entire program `e` (if `e` is a checked form, such as a `λ`) and (2) expressions not in normal form, such as a `λ` immediately applied to an argument, a recursive function declaration, or a let-binding (assuming the rule for `λ` immediately applied to an argument, a recursive function declaration, or a let-binding). Rules with “more synthesis” — such as a synthesizing version of $\text{if} \ a \ \lambda$ — could be added along the lines of previous bidirectional type systems (Xi 1998; Dunfield and Krishnaswami 2013).

Following Davies (2005), an annotation can list several types $\overline{\text{A}}$. Rule $\text{SynAnno}$ chooses one of these, backtracking if necessary. Multiple types may be needed if a `λ`-term is checked against intersection type: when checking `(λx. e)` against `(even → even) ∧ (odd → odd)`, the type of `x` will be `even` inside the left subderivation of $\text{Chk} \land$ but `odd` inside the right subderivation. Thus, if we annotate `x` with `even`, the check against `odd → odd` fails; if we annotate `x` with `odd`, the check against `even → even` fails. For a less contrived example, and for a variant annotation form that reduces backtracking, see Dunfield and Pfenning (2004).

In the appendix, we prove that our bidirectional system is sound and complete with respect to our type assignment system:

**Theorem 8 (Bidirectional soundness).** If $\Gamma \vdash e \iff A$ or $\Gamma \vdash e \Rightarrow A$ then $\Gamma \vdash [e] : A$ where `[e]` is `e` with all annotations erased.

**Theorem 9 (Annotatability).** If $\Gamma \vdash e : A$ then:

1. There exists $e_\text{es}$ such that $[e_\text{es}] = e$ and $\Gamma \vdash e_\text{es} \iff A$.
2. There exists $e_\Rightarrow$ such that $[e_\Rightarrow] = e$ and $\Gamma \vdash e_\Rightarrow \Rightarrow A$.

We also prove that the $\Rightarrow$ and $\iff$ judgments are decidable (Theorem 10).
8 Related Work

Datasort refinements. Freeman and Pfenning (1991) introduced datasort refinements with intersection types, defined the refinement restriction (where $A \land B$ is well-formed only if $A$ and $B$ are refinements of the same type), and developed an inference algorithm in the spirit of abstract interpretation. As discussed earlier, the lack of annotations not only makes the types difficult to see, but makes inference prone to finding long, complex types that include accidental invariants.

Davies (2005), building on the type system developed by Davies and Pfenning (2000), used a bidirectional typing algorithm, guided by annotations on redexes. This system supports parametric polymorphism through a front end based on Damas–Milner inference, but—like Freeman’s system—does not support extensible refinements. Davies’s CIDRE implementation (Davies 2013) goes beyond his formalism by allowing a single type to be refined via multiple declarations, but this has no formal basis; CIDRE appears to simply gather the multiple declarations together, and check the entire program using the combined declaration, even when this violates the expected scoping rules of SML declarations.

Datasort refinements were combined with union types and indexed types by Dunfield and Pfenning (2003, 2004), who noticed the expressive power of nominal subsorting, called “invaluable refinement” (Dunfield 2007b, pp. 113, 220–230).

Giving multiple refinement declarations for a single datatype was mentioned early on, as future work: “embedded refinement type declarations” (Freeman and Pfenning 1991, p. 275); “or even . . . declarations that have their scope limited” (Freeman 1994, p. 167); “it does seem desirable to be able to make local datasort declarations” (Davies 2005, p. 245). But the idea seems not to have been pursued.

Logical frameworks. In the logical framework LF (Harper et al. 1993), data is characterized by declaring constructors with their types. In this respect, our system is closer to LF than to ML: LF doesn’t require all of a type’s constructors to be declared together. By itself, LF has no need for inversion principles. However, systems such as Twelf (Pfenning and Schürmann 1999), Delphin (Poswolsky and Schürmann 2009) and Beluga (Pientka and Dunfield 2010) use LF as an object-level language but also provide meta-level features. One such feature is coverage (exhaustiveness) checking, which needs inversion principles for LF types. Thus, these systems mark a type as frozen when its inversion principle is applied (to process $\%\text{covers}$ in Twelf, or a case expression in Beluga); they also allow the user to mark types as frozen. These systems lack subtyping and subsorting; once a type is frozen, it is an error to declare a new constructor for it.

Lovas (2010) extended LF with refinements and subsorting, and developed a constraint-based algorithm for signature checking. This work did not consider meta-level features such as coverage checking, so it yields no immediate insights about inversion principles or freezing. Since Lovas’s system takes the subsorting relation directly from declarations, rather than by inferring it from a grammar, it supports what Dunfield (2007b) called invaluable refinements; see Lovas’s example (Lovas 2010, pp. 145–147).
Indexed types and refinement types. As the second generation of datasort refinements (exemplified by the work of Davies and Pfenning) began, so did a related approach to lightweight type-based verification: indexed types or limited dependent types \cite{Xi:1999:Partially:Complete:Types, Xi:1998:Dependent:Type:Theory}, in which datatypes are refined by indices drawn from a (possibly infinite) constraint domain. Integers with linear inequalities are the standard example of an index domain; another good example is physical units or dimensions \cite{Dunfield:2007:Standard:Deviation}. More recent work in this vein, such as liquid types \cite{Rondon:2008:Abstract:Refinement:Types}, uses “refinement types” for a mechanism close to indexed types.

Datasort refinements have always smelled like a special case of indexed types. At the dawn of indexed types (and the second generation of datasort refinements), the relationship was obscured by datasorts’ “fellow traveller”, intersection types, which were absent from the first indexed type systems, and remain absent from the approaches now called “refinement types”. That is, while datasorts themselves strongly resemble a specific form of indices—albeit related by a partial order (subtyping), rather than by equality—and would thus suggest that indexed type systems subsume datasort refinement type systems, the inclusion of intersection types confounds such a comparison. Intersection types are present, along with both datasorts and indices, in \cite{Dunfield:2003:Combining:Types:and:Subtyping} and \cite{Dunfield:2007:New:Approach:to:Subtyping}; the relationship is less obscured. But no one has given an encoding of types with datasorts into types with indices, intersections or no.

The focus of this paper is a particular kind of extensibility of datasort refinements, so it is natural to ask whether indexed types and (latter-day) refinement types have anything similar. Indexed types are not immediately extensible: both Xi’s DML and Dunfield’s Stardust require that a given datatype be refined exactly once. Thus, a particular list type may carry its length, or the value of its largest element, or the parity of its boolean elements. By refining the type with a tuple of indices, it may also carry combinations of these, such as its length and its largest element. Subsequent uses of the type can leave out some of the indices, but the combination must be stated up front.

However, some of the approaches descended from DML, such as liquid types, allow refinement with a predicate that can mention various attributes. These attributes are declared separately from the datatype; adding a new attribute does not invalidate existing code. Abstract refinement types \cite{Vazou:2013:Abstract:Refinement:Types} even allow types to quantify over predicates.

Setting aside extensibility, datasort refinements can express certain invariants more clearly and succinctly than indexed types (and their descendants).

Program analysis. \cite{Koot:2015:Type:Systems:for:Exception:Analysis} formulate a type system that analyzes where exceptions can be raised, including match exceptions raised by nonexhaustive case expressions. This system appears to be less precise than datasorts, but has advantages typical to program analysis: no type annotations are required.
9 Future Work

Modular refinements. This paper establishes a critical mechanism for extensible refinements, safe signature extension, in the setting of a core language without modules; refinements are lexically scoped. To scale up to a language with modules, we need to ask: what notions of scope are appropriate? For example, a strict \( \lambda \)-calculus interpreter could be refined with a sort \( \text{val} \) of values, while a lazy interpreter could be refined with a sort \( \text{whnf} \) of terms in weak head normal form. If every \( \text{val} \) is a \( \text{whnf} \), we might want to have \( \text{val} \preceq \text{whnf} \). In the present system, these two refinements could be in separate \texttt{declare} blocks; in that case, \( \text{val} \) and \( \text{whnf} \) could not both be in scope, and the subsorting is not well-formed. Alternatively, one \texttt{declare} block could be nested inside the other. In that case, \( \text{val} \preceq \text{whnf} \) could be given in the nested block, since it would not add new subsortings within the outer refinement. In a system with modules, we would likely want to have \( \text{val} \preceq \text{whnf} \), at least for clients of both modules; such backpatching is currently not allowed, but should be safe since the new subsorting crosses two independent signature blocks (the block declaring \( \text{val} \) and the block declaring \( \text{whnf} \)) without changing the subsortings within each block.

Type polymorphism. Standard parametric polymorphism is absent in this paper, but it should be feasible to follow the approach of \cite{Davies2005}, as long as the unrefined datatype declarations are not themselves extensible (which would break signature well-formedness, even without polymorphism).

Datasort polymorphism. Extensible signatures open the door to sort-bounded polymorphism. In our current system, a function that iterates over an abstract syntax tree and \( \alpha \)-renames free variables—which would conventionally have the type \( \text{exp} \to \text{exp} \)—must be duplicated, even though the resulting tree has the same shape and the same constructors, and therefore should always produce a tree of the same sort as the input tree (at least, if the free variables are not specified with datasorts). We would like the function to check against a polymorphic type \( \forall \alpha \preceq \text{exp}. \alpha \to \alpha \), which works for any sort \( \alpha \) below \( \text{exp} \).

We would like to reason “backwards” from a pattern match over a polymorphic sort variable \( \alpha \). For example, if a value of type \( \alpha \) matches the pattern \( \text{Plus}(x_1, x_2) \), then we know that \( \text{Plus} : (\alpha_1 \ast \alpha_2) \to \alpha \) for some sorts \( \alpha_1 \) and \( \alpha_2 \). The recursive calls on \( x_1 \) and \( x_2 \) must preserve the property of being in \( \alpha_1 \) and \( \alpha_2 \), so \( \text{Plus}(f x_1, f x_2) \) has type \( \alpha \), as needed. The mechanisms we have developed may be a good foundation for adding sort-bounded polymorphism: the \texttt{intersect} function would need to return a signature, as well as a context and type, so that the constructor typing \( \text{Plus} : (\alpha_1 \ast \alpha_2) \to \alpha \) can be made available.

Implementation. Currently, we have a prototype of a few pieces of the system, including a parser and implementations of the \( \Sigma \text{ sig} \) judgment and the \texttt{intersect} function. Experimenting with these pieces was helpful during the design of the system (and reassured us that the most novel parts of our system can be implemented), but they fall short of a usable implementation.
Bibliography

H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. [https://gforge.inria.fr/frs/download.php/file/10994/tata.pdf], 2008. Release of 18 November 2008.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Principles of Programming Languages, pages 238–252, 1977.

Rowan Davies. Practical Refinement-Type Checking. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110.

Rowan Davies. SML checker for intersection and datasort refinements (pronounced “cider”). [https://github.com/rowandavies/sml-cidre], 2013.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In ICFP, pages 198–208, 2000.

Jana Dunfield. Refined typechecking with Stardust. In Programming Languages meets Program Verification (PLPV ’07), 2007a.

Jana Dunfield. A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University, 2007b. CMU-CS-07-129.

Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In ICFP, 2013. [arXiv:1306.6032 [cs.PL]]

Jana Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In Found. Software Science and Computation Structures (FoSSaCS ’03), pages 250–266, 2003.

Jana Dunfield and Frank Pfenning. Tridirectional typechecking. In Principles of Programming Languages, pages 281–292, 2004.

Tim Freeman. Refinement Types for ML. PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-110.

Tim Freeman and Frank Pfenning. Refinement types for ML. In Programming Language Design and Implementation, pages 268–277, 1991.

Gerhard Gentzen. Untersuchungen über das logische Schließen. Mathematische Zeitschrift, 39:176–210, 405–431, 1934. English translation, Investigations into logical deduction, in M. Szabo, editor, Collected papers of Gerhard Gentzen (North-Holland, 1969), pages 68–131.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Journal of the ACM, 40(1):143–184, 1993.

Andrew Kennedy. Programming languages and dimensions. PhD thesis, University of Cambridge, 1996.

Ruud Koot and Jurriaan Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In
ceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, pages 127–138, 2015.

William Lovas. Refinement Types for Logical Frameworks. PhD thesis, Carnegie Mellon University, 2010. CMU-CS-10-138.

Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In Int’l Conf. Automated Deduction (CADE-16), pages 202–206, 1999.

Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Int’l Joint Conference on Automated Reasoning (IJCAR), pages 15–21, 2010.

Adam Poswolsky and Carsten Schürmann. System description: Delphin—a functional programming language for deductive systems. In Int’l Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP’08), volume 228 of Electronic Notes in Theoretical Computer Science, pages 135–141, 2009.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In Information Processing 83, pages 513–523. Elsevier, 1983. [http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/typesabpara.pdf]

John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Programming Language Design and Implementation, pages 159–169, 2008.

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In European Symp. on Programming, pages 209–228, 2013.

Andrew K. Wright. Simple imperative polymorphism. Lisp and Symbolic Computation, 8(4):343–355, 1995.

Hongwei Xi. Dependent Types in Practical Programming. PhD thesis, Carnegie Mellon University, 1998.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Principles of Programming Languages, pages 214–227, 1999.

Noam Zeilberger. The Logical Basis of Evaluation Order and Pattern-Matching. PhD thesis, Carnegie Mellon University, 2009. CMU-CS-09-122.
A Omitted Definitions

A.1 Pattern complement and intersection

\[ \neg p \] Complement of a pattern under unrefined signature \( \mathcal{U} \)

\[ \neg \emptyset = \emptyset \]
\[ \neg \emptyset = \neg (p_1 \sqcup p_2) = (\neg p_1) \cap (\neg p_2) \]
\[ \neg (x \text{ as } p) = \neg p \quad \text{where constructors}(\mathcal{U}) \setminus \{c\} = \{c_1, \ldots, c_n\} \]
\[ \neg (p_1, p_2) = (\neg p_1, \bot \sqcup (\neg p_2) \]

\[ p_1 \cap p_2 \] Intersection of patterns

\[ \emptyset \cap p = p \cap \emptyset = \emptyset \]
\[ \bot \cap p = p \cap \bot = p \]
\[ c(p_1) \cap c(p_2) = c(p_1 \cap p_2) \]
\[ c_1(p_1) \cap c_2(p_2) = \emptyset \quad \text{where } c_1 \neq c_2 \]
\[ (p_1, p_2) \cap c(p) = c(p) \cap (p_1, p_2) = \emptyset \quad \text{where } p \notin \{\emptyset, \bot \ldots \sqcup \ldots \} \]

Fig. 13. Pattern complement and pattern intersection
A.2 Evaluation contexts, matching, and stepping

Evaluation contexts \( E \) := \([\ ] | E \ e | v E | c E | (E, e) | (v, E) | \text{case } E \) of \( ms \)

\[
\begin{array}{ll}
\text{p match } v \rightarrow \theta & \text{Value } v \text{ matches pattern } p \text{ by substitution } \theta, \text{i.e. } [\theta]p = v \\
\overrightarrow{\text{match } v \rightarrow \cdot} \text{MatchWild} & p \text{ match } v \rightarrow \theta \\
\text{MatchAs} & x \text{ as } p \text{ match } v \rightarrow \theta, v/x \\
\text{MatchOr} & p \text{ match } v \rightarrow \theta \\
\text{MatchCon} & c(p) \text{ match } c(v) \rightarrow \theta \\
\text{MatchPair} & (p_1, p_2) \text{ match } (v_1, v_2) \rightarrow (\theta_1, \theta_2) \\
\text{MatchUnit} & \phi \text{ match } \cdot \\
\text{NoMWild} & \text{NoMAs} \\
\text{NoMOr} & \text{NoMAss} \\
\text{NoMPairHead} & (p_1, p_2) \text{ match } v \rightarrow (\theta_1, \theta_2) \\
\text{NoMPairInner} & \text{NoMPairUnit} \\
\text{NoMConHead} & \text{NoMConInner} \\
\text{NoMUnit} & v \neq () \\
\end{array}
\]

Fig. 14. Operational semantics
A.3 Substitution typing

\[ \Sigma; \Gamma \vdash \theta : \Gamma' \]

Substitution \( \theta \), applied to something well-formed under \( \Sigma; (\Gamma, \Gamma') \), replaces variables in \( \Gamma' \) to yield something well-formed under \( \Gamma \)

\[ \Sigma; \Gamma \vdash \cdot : \cdot \]

SubstEmpty

\[ \Sigma; \Gamma \vdash \theta : \Gamma' \]

\[ \Sigma; \Gamma \vdash \nu : A \]

SubstVar

Fig. 15. Substitution typing

Substitution typing (Figure 15) is used to state Lemma 15 (Multiple substitution).

B Proofs

B.1 Properties of subtyping

Lemma 6 (Reflexivity). If \( \Sigma \vdash A \) type then \( \Sigma \vdash A \leq A \).

Proof. By structural induction on \( A \).

The case for \( A = 1 \), the case for \( A = A_1 \cdot A_2 \), and the case for \( A = A_1 \rightarrow A_2 \) are straightforward.

- Case \( A = s \): Use \( \leq \text{Refl} \) and \( \leq \text{Data} \).

Lemma 7 (Transitivity).

If \( \Sigma \vdash A \leq B \) and \( \Sigma \vdash B \leq C \) where \( \Sigma \vdash A \) type and \( \Sigma \vdash B \) type and \( \Sigma \vdash C \) type then \( \Sigma \vdash A \leq C \).

Proof. By simultaneous induction on the given derivations.

- If either derivation is by \( \leq \text{1} \), then \( A = B \) or \( B = C \), and the other given derivation is the desired result.
- If both derivations are by \( \leq \text{a} \), apply the i.h. as needed, then apply \( \leq \text{<} \).
- If both derivations are by \( \leq \text{e} \), apply the i.h. as needed, then apply \( \leq \text{<=} \).
- If both derivations are by \( \leq \text{data} \), apply \( \leq \text{Trans} \), then apply \( \leq \text{Data} \).
- If the first derivation is by \( \leq \text{\&L}_k \) we have \( A = A_1 \land A_2 \).
  \[ \Sigma \vdash A_k \leq B \quad \text{Subderivation} \]
  \[ \Sigma \vdash A_k \leq C \quad \text{By i.h.} \]
  \[ \Sigma \vdash (A_1 \land A_2) \leq C \quad \text{By } \leq \text{\&L}_k \]
- If the second derivation is by \( \leq \text{\&R} \) we have \( C = C_1 \land C_2 \).
  \[ \Sigma \vdash A \leq B \quad \text{Given} \]
  \[ \Sigma \vdash B \leq C_1 \quad \text{Subderivation} \]
  \[ \Sigma \vdash A \leq C_1 \quad \text{By i.h.} \]
  \[ \Sigma \vdash A \leq C_2 \quad \text{Similar (} B \leq C_2 \text{ subderivation)} \]
  \[ \Sigma \vdash A \leq (C_1 \land C_2) \quad \text{By } \leq \text{\&R} \]
If the first derivation $D_1$ is by $\leq \land R$ and the second derivation $D_2$ is by $\leq \land L_k$, we have $B = B_1 \land B_2$.

- $\Sigma \vdash A \leq B_k$ Subderivation of $D_1$
- $\Sigma \vdash B_k \leq C$ Subderivation of $D_2$
- $\Sigma \vdash A \leq C$ By i.h.

Other combinations of concluding rules are impossible. □

B.2 Subsort properties

Theorem 1 (Non-adjacent preservation).
If $\Sigma_2$ preserves subsorting of $\Sigma_1$ and $\Sigma_3$ preserves subsorting of $\Sigma_1$ and $\Sigma_2$ and $\Sigma_3$ are non-adjacent then $\Sigma_3$ preserves subsorting of $(\Sigma_1, \Sigma_2)$.

Proof. Suppose $\Sigma_1, \Sigma_2, \Sigma_3 \vdash s \preceq t$ where $s, t \in \text{dom}(\Sigma_1, \Sigma_2)$. Following Definition 1, we need to show that $\Sigma_1, \Sigma_2 \vdash s \preceq t$.

Every derivation of a subsorting judgment is essentially a finite path in a directed graph from the subsort to the supersort. The path from $s$ to $t$ must pass through edges (subsortings) in $\Sigma_3$ a finite number of times, say $n$ times. Proceed by induction on $n$:

- If $n = 0$, the path from $s$ to $t$ does not pass through $\Sigma_3$ at all, so we can simply replace $(\Sigma_1, \Sigma_2, \Sigma_3)$ in the given derivation with $(\Sigma_1, \Sigma_2)$.
- If $n > 0$, choose the last $\Sigma_3$ segment in the path:

\[
\begin{array}{c}
s \preceq \cdots \preceq s_1 \preceq \underbrace{t_3 \preceq \cdots \preceq t'_3}_{\text{subsort edges in } \Sigma_3} \preceq s'_1 \preceq \cdots \preceq t
\end{array}
\]

Here, $t_3, t'_3 \in \text{dom}(\Sigma_3)$. Now consider the vertices (sorts) $s_1$ and $s'_1$. These sorts must be in $\text{dom}(\Sigma_1, \Sigma_2)$. Since $\Sigma_2$ and $\Sigma_3$ are non-adjacent, neither $s_1$ nor $s'_1$ can be in $\text{dom}(\Sigma_2)$. Therefore, $s_1, s'_1 \in \text{dom}(\Sigma_1)$.

All the edges from $t_3$ to $t'_3$ are in $\Sigma_3$, so from $\Sigma_1, \Sigma_2, \Sigma_3 \vdash s_1 \preceq s'_1$ we get $\Sigma_1, \Sigma_3 \vdash s_1 \preceq s'_1$.

It is given that $\Sigma_3$ preserves subsorting of $\Sigma_1$. Therefore, $\Sigma_1 \vdash s_1 \preceq s'_1$, yielding a path

\[
\begin{array}{c}
s \preceq \cdots \preceq s_1 \preceq \underbrace{s'_1 \preceq \cdots \preceq t}_{\text{subsort edges in } \Sigma_1}
\end{array}
\]

This path has one less $\Sigma_3$ segment than the one we started with, so the result follows by induction. □

B.3 Strengthening, weakening, and substitution

Weakening of the supporting judgments

Lemma 8 (Weakening (lowest level)). Given $\Sigma$ and $\Sigma'$ such that $\Sigma' \leq_{\text{sig}} \Sigma$: 
Suppose that $t$. Proof. $\Sigma, \Omega, \Sigma$ then and $S$

(ii). part (ii) of the i.h.

Proof. For each part, by induction on the given derivation, assuming lower-numbered parts.

Part (i): All cases are straightforward.

Part (ii), subsorting: All 3 cases $\leq$Assum, $\leq$Refl and $\leq$Trans are straightforward.

Part (iii), subtyping: All cases are straightforward; the case for $\leq$Data uses part (ii) of the i.h.

(ii).

Part (v), constructor typing: There is one case, $\text{ConArr}$ which uses the i.h. (ii).

Part (vi), constructor type well-formedness: use part (i).

$\square$

Lemma 9 (Weakening ‘SafeConAt’).

If $\Sigma, \Sigma'; S(K) \vdash c : A \rightarrow s$ safe at $t$ then $\Sigma, \Omega, \Sigma'; S(K) \vdash c : A \rightarrow s$ safe at $t$.

Proof. By inversion on $\text{SafeConAt}$ and applying Lemma 8 (Weakening (lowest level)) parts (ii), (ii), and (iii), and then applying $\text{SafeConAt}$. $\square$

Lemma 10 (Supersorting ‘SafeConAt’).

If $\Sigma; S(K) \vdash c : A \rightarrow s$ safe at $t'$ and $\Sigma, S(K) \vdash t' \leq t$ then $\Sigma; S(K) \vdash c : A \rightarrow s$ safe at $t$.

Proof. Some of the premises of $\text{SafeConAt}$ do not involve $t'$ at all, so we can reuse them directly. The exception is the premise $\Sigma, S(K) \vdash s' \leq t'$. Applying $\leq$Trans gives $\Sigma, S(K) \vdash s' \leq t$. Now we can apply $\text{SafeConAt}$. $\square$

Theorem 2 (Weakening ‘safe’).

If $(\Sigma, \Sigma')$ sig and $(\Sigma, \Omega)$ sig and $\text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset$ and $\text{dom}(\Sigma, \Omega, \Sigma') \cap S = \emptyset$ and $K$ does not mention anything in $\text{dom}(\Omega)$ and $S(K)$ preserves subsorting for $(\Sigma, \Sigma')$

and $(c : A \rightarrow s) \in K$ and $\Sigma, \Sigma'; S(K) \vdash c : A \rightarrow s$ safe

then $\Sigma, \Omega, \Sigma'; S(K) \vdash c : A \rightarrow s$ safe.

Proof.

1. $s \in S$ Given

2. $\Sigma, \Sigma', S(K) \vdash c : A \rightarrow s$ contype $\text{BlockCon}$

$\Sigma, \Sigma', S(K) \vdash c : A \rightarrow s$ contype $\text{vi}$

$\Sigma, \Sigma', S(K) \vdash c : A \rightarrow s$ safe at $t$ $\text{vi}$ for all $t \in \text{dom}(\Sigma, \Sigma')$ such that ...

Suppose that $t \in \text{dom}(\Sigma, \Omega, \Sigma')$ and $\Sigma, \Omega, \Sigma', S(K) \vdash s \leq t$. 


– **Case** \( t \in \text{dom}(\Sigma, \Sigma') \):

\[ \Sigma, \Omega, \Sigma', S(K) \vdash s \preceq t \quad \text{Assumption} \]

It is given that \( S(K) \) preserves subsorting for \( (\Sigma, \Sigma') \). By inversion on \( (\Sigma, \Sigma') \) sig, signature \( \Sigma' \) preserves subsorting for \( \Sigma \). Combining these, we have that \( (\Sigma', S(K)) \) preserves subsorting for \( \Sigma \).

It is also given that \( K \) does not mention anything in \( \text{dom}(\Omega) \), and by inversion on \( (\Sigma, \Omega) \) sig (and using \( \text{dom}(\Omega, \Sigma') = \emptyset \)), we know that \( \Omega \) does not mention anything in \( S \); therefore, \( S(K) \) and \( \Omega \) are non-adjacent (Definition 2).

By Theorem 1 with \( \Sigma_1 = \Sigma \) and \( \Sigma_2 = (\Sigma', S(K)) \) and \( \Sigma_3 = \Omega \), we have that \( \Omega \) preserves subsorting of \( (\Sigma, \Sigma', S(K)) \).

Preservation of subsorting is invariant under signature permutation, so \( \Omega \) can be permuted leftward: \( \Sigma, \Omega, \Sigma', S(K) \vdash s \preceq t \) if and only if \( \Sigma, \Sigma', S(K) \vdash s \preceq t \), and we assumed the former judgment. Therefore:

\[ \Sigma, \Sigma', S(K) \vdash s \preceq t \]

This is the guard of one of the premises under the above “for all”. Therefore:

\[ \Sigma, \Omega, \Sigma', S(K) \vdash c : A \to s \text{ safe at } t \quad \text{By Lemma 9 (Weakening ‘SafeConAt’) \[1\]} \]

– **Case** \( t \in \text{dom}(\Omega) \):

Here we have \( \Sigma, \Omega, \Sigma', S(K) \vdash s \preceq t \).

It is given that \( K \) does not mention any sorts in \( \text{dom}(\Omega) \). Therefore, \( s \preceq t \) must have been derived transitively: there must exist another sort \( s' \in \text{dom}(\Sigma, \Sigma') \cup S \) such that \( s \preceq s' \) and \( s' \preceq t \). Using the reasoning in the subcase for when \( t \in \text{dom}(\Sigma, \Sigma') \), we get

\[ \Sigma, \Omega, \Sigma', S(K) \vdash c : A \to s \text{ safe at } t \]

\[ \Sigma, \Omega, \Sigma', S(K) \vdash c : A \to s \text{ safe at } t \quad \text{By Lemma 10 (Supersorting ‘SafeConAt’) \[10\]} \]

This shows the “for all” part of \( \text{BlockCon} \). Together with “1” and “2” above, we can apply \( \text{BlockCon} \):

\[ \Sigma, \Omega, \Sigma', S(K) \vdash c : A \to s \text{ safe at } t \quad \text{By BlockCon} \]

**Theorem 3 (Signature Interleaving).**

If \( (\Sigma, \Sigma') \) sig and \( (\Sigma, \Omega) \) sig and \( \text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset \) then \( (\Sigma, \Omega, \Sigma') \) sig.

**Proof.** By induction on \( \Sigma' \).

If \( \Sigma' = \cdot \), then we already have our result.

Otherwise, \( \Sigma' = (\Sigma_0', S(K)) \).
We still need to prove the following:

\[ (\Sigma, \Sigma', S(K)) \ \text{sig} \]

\[ \text{S} \cap \text{dom}(\Sigma, \Sigma') = \emptyset \quad \text{By inversion on \textbf{SigBlock}} \]

\[ (\Sigma, \Sigma', S(K)) \vdash \Sigma' = (\Sigma, \Sigma', S(K)) \quad \text{[Preservation]} \]

\[ (\Sigma, \Sigma'; S(K)) \vdash c : A \rightarrow s \ \text{safe} \quad \text{for all } (c : A \rightarrow s) \in K \]

1

\[ \text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset \quad \text{Given} \]

\[ \text{dom}(\Sigma', S(K)) \cap \text{dom}(\Omega) = \emptyset \quad \Sigma' = (\Sigma', S(K)) \]

\[ (\text{dom}(\Sigma') \cup \text{S}) \cap \text{dom}(\Omega) = \emptyset \quad \text{By def. of } \text{dom}(-) \]

\[ \text{S} \cap \text{dom}(\Omega) = \emptyset \quad \text{By a property of } \cap \text{ and } \bigcup \]

\[ \text{S} \cap \text{dom}(\Sigma, \Sigma') = \emptyset \quad \text{Above} \]

2

\[ \text{S} \cap \text{dom}(\Sigma, \Omega, \Sigma') = \emptyset \quad \text{By def. of } \text{dom}(-) \text{ and a property of } \cap \]

We still need to prove the following:

3

\[ (\Sigma, \Omega, \Sigma'_0) \vdash \Sigma = (\Sigma, \Omega, \Sigma'_0) \quad \text{To be proved} \]

4

\[ \Sigma, \Omega, \Sigma'_0; S(K) \vdash c : A \rightarrow s \ \text{safe} \quad \text{To be proved for all } (c : A \rightarrow s) \in K \]

\[ (\Sigma, \Omega, \Sigma'_0, S(K)) \ \text{sig} \quad \text{By \textbf{SigBlock} (1, 2, 3, 4)} \]

---

**Proof of 3:**

We have \((\Sigma, \Omega)\) sig.

Elaborating equation 3, we need to show, for all \(t_1, t_2 \in \text{dom}(\Sigma, \Omega, \Sigma'_0)\), that each direction holds:

- (a) If \(\Sigma, \Omega, \Sigma'_0 \vdash t_1 \equiv t_2\) then \(\Sigma, \Omega, \Sigma'_0, S(K) \vdash t_1 \equiv t_2\).
- (b) If \(\Sigma, \Omega, \Sigma'_0, S(K) \vdash t_1 \equiv t_2\) then \(\Sigma, \Omega, \Sigma'_0 \vdash t_1 \equiv t_2\).

For direction (a), Lemma 8 [Weakening (lowest level)] (ii) suffices.

For direction (b):

We have \((\Sigma, \Sigma'_0, S(K))\) sig.

By inversion on \textbf{SigBlock} the block K preserves subsorting of the signature \((\Sigma, \Sigma'_0)\).

By inversion on \textbf{BlockSubsort} none of the subsortings added by K are in \text{dom}(\Omega), and none of the subsortings added by \Omega are in S. That is, S(K) and \Omega are non-adjacent.

By Theorem 1 with \(\Sigma_1 = (\Sigma, \Sigma'_0)\) and \(\Sigma_2 = \Omega\) and \(\Sigma_3 = S(K)\), we have that \(\Sigma_3\) preserves subsorting of \((\Sigma_1, \Sigma_2)\), that is, \(S(K)\) preserves subsorting of \((\Sigma, \Sigma'_0, \Omega)\).

Preservation of subsorting is invariant under signature permutation, so \(S(K)\) preserves subsorting of \((\Sigma, \Omega, \Sigma'_0)\). By Definition 1 (b) holds.

---

**Proof of 4:**

Most of the work will be done by Theorem 2. We need to show all of that theorem’s conditions. All of our meta-variables match up with the statement of the lemma, except that our \(\Sigma'_0\) will play the role of \(\Sigma'\).
• \((\Sigma, \Sigma')\) \textit{sig.} Above.
• \((\Sigma, \Omega)\) \textit{sig.} Given.
• \(\text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset\): Follows from \(\text{dom}(\Sigma') \cap \text{dom}(\Omega) = \emptyset\), which was given.
• \(\text{dom}(\Sigma, \Omega, \Sigma') \cap S = \emptyset\): This is “2”, shown above.
• \(K\) does not mention anything in \(\text{dom}(\Omega)\):
  - If \(K\) mentioned anything in \(\text{dom}(\Omega)\), it would contradict the premise of \textit{BlockSubsort} and/or the \textit{contype} premise of \textit{BlockCon}.
• \(K\) preserves subsorting for \((\Sigma, \Sigma')\): This is “Preservation”, shown above.
• \((c : A \rightarrow s) \in K\): Assumption.
• \(\Sigma, \Sigma', \Sigma(K) \vdash c : A \rightarrow s\) \textit{safe}: Above.

The judgment marked 4 follows by Theorem 2.

Now we can apply \textit{SigBlock} to
\[(\Sigma, \Omega, \Sigma', \Sigma(K)) \vdash \text{sig} \quad \text{By \textit{SigBlock} (1, 2, 3, 4)} \]

**Intersect strengthening**

\textbf{Lemma 11} (Properties of stronger contexts).

(i) Reflexivity: For all contexts \(\Gamma\), we have \(\Sigma \vdash \Gamma \subseteq \text{ctx} \Gamma\).
(ii) Transitivity: If \(\Sigma \vdash \Gamma'' \subseteq \text{ctx} \Gamma'\) and \(\Sigma \vdash \Gamma' \subseteq \text{ctx} \Gamma\) then \(\Sigma \vdash \Gamma'' \subseteq \text{ctx} \Gamma\).
(iii) Concatenation: If \(\Sigma \vdash \Gamma_1' \subseteq \text{ctx} \Gamma_1\) and \(\Sigma \vdash \Gamma_2' \subseteq \text{ctx} \Gamma_2\) then \(\Sigma \vdash (\Gamma_1', \Gamma_2') \subseteq \text{ctx} (\Gamma_1, \Gamma_2)\).

\textit{Proof.} Part (i): By induction on \(\Gamma\), using Lemma 6 \textit{Reflexivity}.
Part (ii): Use Lemma 7 \textit{Transitivity}.
Part (iii): By induction on \(\Gamma_2\). \(\square\)

\textbf{Lemma 12} (Constructor).

If \(\Sigma\) \textit{sig} and \(\Sigma^+ \text{ sig} and \Sigma^+ \leq_{\text{sig}} \Sigma\)
and \((c : A_c^+ \rightarrow s_c^+) \in \Sigma^+\) and \(\Sigma^+ \vdash s_c^+ \leq s\) and \(s \in \text{dom}(\Sigma)\)
then \((c : A_c \rightarrow s_c) \in \Sigma\) and \(\Sigma^+ \vdash s_c^+ \leq s_c\) and \(A_c^+ \leq A_c\).

\textit{Proof.} If \((c : A_c^+ \rightarrow s_c^+) \in \Sigma\), then: Let \(A_c = A_c^+\) and \(s_c = s_c^+\). By Lemma 6 \textit{Reflexivity}, \(\Sigma^+ \vdash A_c^+ \leq A_c\).

Otherwise, by inversion on \(\Sigma^+ \text{ sig}\), there is a derivation using \textit{BlockCon}
that says that, for some \(\Sigma_1\) such that \(\Sigma = (\Sigma_1, \Sigma_2)\), we have \(\Sigma_1; \Sigma(K) \vdash c : A_c^+ \rightarrow s_c^+\) \textit{safe at} \(t\) for all \(t \in \text{dom}(\Sigma_1)\) such that \(\Sigma_1; \Sigma(K) \vdash s_c^+ \leq t\).

We have \(\Sigma^+ \vdash s_c^+ \leq s\). By well-formedness of the given signatures, all blocks preserve subsorting of previously-defined sorts. Therefore

\[\Sigma_1; \Sigma(K) \vdash c : A_c^+ \rightarrow s_c^+ \text{ safe at } s\]
Lemma 13 (Intersect strengthening).
If $\Sigma_+ \text{ sig}$ and $\Sigma \text{ sig}$ and $\Sigma_+ \leq_{\text{sig}} \Sigma$ and $\Sigma \vdash A$ type
and $\Sigma_+ \vdash A_+ \leq A$
then $\Sigma_+ \vdash \text{intersect}(\Sigma_+ \vdash A_+; p) \leq_{\text{trk}} \text{intersect}(\Sigma \vdash A; p)$.

Proof. By structural induction on $p$.
Suppose we have
$$\text{intersect}(\Sigma_+ \vdash A_+; p) = \{(\Sigma^1_+; \Gamma^1_+ \vdash B^1_+), \ldots, (\Sigma^n_+; \Gamma^n_+ \vdash B^n_+)\}$$
$$\text{intersect}(\Sigma \vdash A; p) = \{(\Sigma^1; \Gamma^1 \vdash B^1), \ldots, (\Sigma^m; \Gamma^m \vdash B^m)\}$$
For each track $B^*_+ = (\Sigma^*_+; \Gamma^*_+ \vdash B^*_+)$ in $\text{intersect}(\Sigma \vdash A; p)$, we need to find
some track $B^*$ in $\text{intersect}(\Sigma_+ \vdash A_+; p)$ such that $B^*_+ \leq_{\text{trk}} B^*$.

- Case $p = \_$. In this case, $\text{intersect}$ ignores the signature entirely. The result follows by Lemma 11 (Properties of stronger contexts) (i), and the given subtyping $\Sigma_+ \vdash A_+ \leq A$.
- Case $p = (x$ as $p_0)$:
  $\text{intersect}(\Sigma_+ \vdash A_+; p_0) \leq_{\text{trk}} \text{intersect}(\Sigma \vdash A; p_0)$ By i.h.
Suppose $(\Gamma'_+, x : B_+ \vdash B_+) \in \text{intersect}(\Sigma_+ \vdash A_+; x$ as $p_0)$ By definition of intersect
  $$(\Gamma' + B) \in \text{intersect}(\Sigma \vdash A; p_0)$$
  $$(\Gamma'_+ \vdash B_+) \leq_{\text{trk}} (\Gamma' \vdash B)$$
  $$\Sigma \vdash \Gamma'_+ \leq_{\text{ctx}} \Gamma'$$
  $$\Sigma_+ \vdash B_+ \leq B$$
  $$(\Gamma'_+, x : B + B) \in \text{intersect}(\Sigma \vdash A; x$ as $p_0)$$ By definition of intersect
  $$(\Gamma'_+ \vdash B_+) \leq_{\text{trk}} (\Gamma'_+, x : B \vdash B)$$
  $$(\Gamma'_+ \vdash B_+) \leq_{\text{trk}} (\Gamma'_+, x : B + B)$$

- Case $p = \emptyset$. In this case, $\text{intersect}$ ignores the signature—and the result is the empty set, which is, trivially, a strengthening of the empty set.
- Case $p = (p_1 \cup p_2)$:
  By i.h., $\text{intersect}(\Sigma_+ \vdash A_+; p_1) \leq_{\text{trk}} \text{intersect}(\Sigma \vdash A; p_1)$.
Similarly, $\text{intersect}(\Sigma_+ \vdash A_+; p_2) \leq_{\text{trk}} \text{intersect}(\Sigma \vdash A; p_2)$.
Then:
$$\text{intersect}(\Sigma_+ \vdash A_+; p) \leq_{\text{trk}} (\text{intersect}(\Sigma_+ \vdash A_+; p_1) \cup \text{intersect}(\Sigma_+ \vdash A_+; p_2))$$
- Case \( p = (p_1 \ast p_2) \):

\[
\begin{align*}
\Sigma_+ & \vdash (A_+^1 \ast A_+^2) \leq (A^1 \ast A^2) \quad \text{Given} \\
\Sigma_+ & \vdash A_+^1 \leq A^1 \quad \text{By inversion on } \leq
\end{align*}
\]

\( \Gamma \) intersect \( \Sigma_+ \vdash A_+^1 ; p_1 \) \( \leq_{\text{trk}} \) intersect \( \Sigma \vdash A^1 ; p_1 \) \( \leq_{\text{trk}} \) By i.h.

\( \Gamma \vdash A_+^2 \) \( \leq_{\text{trk}} \) intersect \( \Sigma \vdash A^2 ; p_2 \) \( \leq_{\text{trk}} \) By i.h.

Suppose:

\[
\begin{align*}
(\Gamma_1^1 \vdash B_1^1) & \in \text{intersect}(\Sigma_+ \vdash A_+^1 ; p_1) \\
(\Gamma_1^2 \vdash B_1^2) & \in \text{intersect}(\Sigma_+ \vdash A_+^2 ; p_2)
\end{align*}
\]

\( (\Gamma_1^1, \Gamma_1^2 \vdash B_1^1 \ast B_1^2) \in \text{intersect}(\Sigma_+ \vdash A_1 \ast A_2 ; (p_1, p_2)) \) By definition of intersect

\[
\begin{align*}
(\Gamma^1 \vdash B^1) & \in \text{intersect}(\Sigma \vdash A_1 ; p_1) \quad \text{By above } \leq_{\text{trk}} \\
(\Gamma_1^1 \vdash B_1^1) & \leq_{\text{trk}} (\Gamma^1 \vdash B^1) \quad " \\
\Sigma_+ & \vdash \Gamma_1^1 \leq_{\text{ctx}} \Gamma^1 \quad \text{From Definition 5} \\
\Sigma_+ & \vdash B_1^1 \leq B^1 \quad " \\
\Sigma_+ & \vdash \Gamma_1^2 \leq_{\text{ctx}} \Gamma^2 \quad \text{Similar, with 2 substituted for 1} \\
\Sigma_+ & \vdash B_1^2 \leq B^2 \quad " \\
\Sigma_+ & \vdash (\Gamma_1^1, \Gamma_1^2) \leq_{\text{ctx}} (\Gamma^1, \Gamma^2) \quad \text{By Lemma 11 (iii)} \\
\Sigma_+ & \vdash (B_1^1 \ast B_1^2) \leq (B^1 \ast B^2) \quad \text{By } \leq_1
\end{align*}
\]

\( (\Gamma^1, \Gamma^2 \vdash (B^1 \ast B^2)) \in \text{intersect}(\Sigma \vdash A_1 \ast A_2 ; (p_1, p_2)) \) By definition of intersect

\( (\Gamma_1^1, \Gamma_1^2 \vdash B_1^1 \ast B_1^2) \leq_{\text{trk}} (\Gamma^1, \Gamma^2 \vdash B^1 \ast B^2) \) By Definition 5

- Case \( p = c(p_0) \): Suppose \( c : (A_+^c \rightarrow s_+^c) \in \Sigma_+ \) where \( \Sigma_+ \vdash s_+^c \preceq s \).

\[
\begin{align*}
(c : A_+^c \rightarrow s_+^c) & \in \Sigma_+ \quad \text{Above} \\
\Sigma_+ & \vdash s_+^c \preceq s \quad \text{Above} \\
\Sigma & \vdash s \text{ type} \quad \text{Given} \\
s & \in \text{dom}(\Sigma) \quad \text{By inversion on } \text{WfTypeSort} \\
\Sigma_+ & \vdash s_+^c \preceq s \quad \text{"} \\
\Sigma & \vdash s_+^c \quad \text{"} \\
(c : A_+^c \rightarrow s_+^c) & \in \Sigma \quad \text{By Lemma 12} \quad \text{Constructor} \\
\Sigma_+ & \vdash s_+^c \preceq s_+^c \quad \text{"} \\
\Sigma_+ & \vdash A_+^c \leq A_+^c \quad \text{"}
\end{align*}
\]

\( \Gamma_1^1 \vdash A_1^c ; p_0 \) \( \leq_{\text{trk}} \) intersect \( \Sigma \vdash A_1^c ; p_0 \) \( \leq_{\text{trk}} \) By i.h.

Suppose \( (\Gamma_1^1 \vdash B_1^c) \in \text{intersect}(\Sigma_+ \vdash A_+^c ; p_0) \).
(Γ′ ⊢ B′) ∈ intersect(Σ ⊢ A; p₀)  
By above ≤trk
(Γ⁺ ⊢ B⁺) ≤trk (Γ′ ⊢ B′) "
Σ⁺ ⊢ Γ⁺ ≤ctx Γ′  
By Definition 5
Σ ⊢ B⁺ ≤ B′ "
Σ⁺ ⊢ Γ⁺ ≤ctx Γ′  
Above
Σ⁺ ⊢ s⁺ ≤ s⁺  
Above
Σ⁺ ⊢ s⁺ ≤ s⁺  
By ≤Data
Σ⁺ ⊢ (Γ⁺ ⊢ s⁺) ≤trk (Γ′ ⊢ s⁺)  
By Definition 5 □

Weakening of the main judgments

Theorem 4 (Weakening).
If (Σ, Σ') sig, (Σ, Ω) sig, dom(Σ') ∩ dom(Ω) = ∅ and Σ, Ω, Σ′ ⊢ Γ⁺ ≤ctx Γ then
(1) If Σ, Σ', Γ ⊢ e : A then Σ, Ω, Σ'; Γ⁺ ⊢ e : A.
(2) If Σ, Σ', Γ ⊢ θ : Γ' then Σ, Ω, Σ'; Γ⁺ ⊢ θ : Γ'.
(3) If Σ, Σ', Γ; p : A ⊢ ms : D then Σ, Ω, Σ'; Γ⁺; p : A ⊢ ms : D.

Proof. For each part, by induction on the height of the given derivation.
Part (1), expression typing:
- τVar apply τVar
- Sub apply the i.h. (1) and Lemma 8 (Weakening (lowest level)) (iii).
- τ[Γ⁺]→E E Data straightforward. The Data case uses Lemma 8 (Weakening (lowest level)) (iv).
- Case Σ, Σ', Γ ⊢ e : B  
  Σ, Σ', Γ⁺ ⊢ e : B  
  Subderivation
  Σ, Ω, Σ⁺; Γ ⊢ e : B  
  By i.h. (1)
  Σ, Ω, Σ⁺; Γ⁺ ⊢ e : B  
  By i.h. (1)
Σ, Ω, Σ⁺; Γ⁺ ⊢ (case e of ms) : A  
By DataE
- Case (Σ, Σ', Σ'') sig  
  Σ, Σ', Γ ⊢ A type  
  Σ, Σ', Σ''; Γ ⊢ e₀ : A  
  Declare
  Σ, Σ', Γ⁺ ⊢ (declare Σ'' in e₀) : A  
  Subderivation
  (Σ, Ω) sig  
  Given
  dom(Σ') ∩ dom(Ω) = ∅  
  Given
  dom(Σ'') ∩ dom(Ω) = ∅  
  By renaming dom(Σ'')
  (dom(Σ') ∪ dom(Σ'')) ∩ dom(Ω) = ∅  
  By set theory
  dom(Σ', Σ'') ∩ dom(Ω) = ∅  
  By def. of dom(−)
1. \((\Sigma, \Omega, \Sigma') \ sig\) By Theorem 3
   \(\Sigma, \Sigma' \vdash A \ type\) Subderivation

2. \(\Sigma, \Omega, \Sigma' \vdash A \ type\) By Lemma 8 (Weakening (lowest level)) (i)
   \(\Sigma, \Sigma', \Sigma'' ; \Gamma \vdash e_0 : A\) Subderivation
3. \(\Sigma, \Omega, \Sigma', \Sigma'' ; \Gamma \vdash e_0 : A\) By i.h. (1)
   \(\Sigma, \Omega, \Sigma'; \Gamma \vdash (\text{declare } \Sigma'' \text{ in } e_0) : A\) By Declare on 1, 2, 3

Part (2), substitution typing: In the SubstVar case, use part (1).
Part (3), where a derivation of \(\Sigma, \Sigma' ; \Gamma ; p : B \vdash ms : D\) is given:

- Case \(\text{intersect}(\Sigma, \Sigma' ; A; p) = \emptyset\)
  \(\Sigma, \Sigma' ; \Gamma ; p : A \vdash \emptyset : D\) TypeMsEmpty
  Suppose \(\text{intersect}(\Sigma, \Omega, \Sigma' ; A^+; p) \neq \emptyset\).
  By Lemma 13 (Intersect strengthening), for each track in \(\text{intersect}(\Sigma, \Omega, \Sigma' ; A^+; p)\) there exists a track in \(\text{intersect}(\Sigma, \Sigma' ; A; p)\). But we have as a premise that \(\text{intersect}(\Sigma, \Sigma' ; A; p) = \emptyset\), a contradiction.
  Therefore \(\text{intersect}(\Sigma, \Omega, \Sigma' ; A^+; p) = \emptyset\).
  The result follows by TypeMsEmpty.

- Case for all \((\Gamma \vdash B)\)
  \(\Sigma, \Sigma', A \sqsubseteq \tau\) \(\in \text{intersect}(\Sigma \vdash A; p \cap p_1)\):
  \(U \vdash p_1 : \tau\) \(\Sigma, \Gamma, \Gamma', e_1 : D\) \(\Sigma, \Sigma'; \Gamma; (p \cap \neg p_1) : A \vdash ms : D\) TypeMs
  \(\Sigma, \Sigma'; \Gamma; p : A \vdash ((p_1 \Rightarrow e_1) \ I ms) : D\).

\(\Sigma, \Sigma' \vdash A \sqsubseteq \tau\) Subderivation
\(\Sigma, \Omega, \Sigma' \vdash A \sqsubseteq \tau\) By Lemma 8 (Weakening (lowest level)) (vi)
\(U \vdash p_1 : \tau\) Subderivation
\((\Sigma, \Omega, \Sigma') \leq_{\text{sig}} (\Sigma, \Sigma')\) By Definition 3
\(\Sigma, \Omega, \Sigma' \vdash \Gamma^+ \leq_{\text{str}} \Gamma\) Given
\(\Sigma, \Sigma' \vdash A \ type\) Given

By Lemma 13
\(\text{intersect}(\Sigma, \Omega, \Sigma' ; A; p \cap p_1) \leq_{\text{trk}} \text{intersect}(\Sigma, \Sigma' ; A; p \cap p_1)\)

By Definition 5 each track in \(\text{intersect}(\Sigma, \Omega, \Sigma' ; A; p \cap p_1)\) is stronger than some track in \(\text{intersect}(\Sigma, \Sigma' ; A; p \cap p_1)\).
That is, if \((\Gamma_0^+ \vdash B^+) \in \text{intersect}(\Sigma, \Omega, \Sigma' ; A; p \cap p_1)\), then
\(\Sigma, \Omega, \Sigma' \vdash (\Gamma_0^+ \vdash B^+) \leq_{\text{trk}} (\Gamma_0 \vdash B)\)
where \((\Gamma_0 \vdash B) \in \text{intersect}(\Sigma, \Sigma' ; A; p \cap p_1)\).
\[\Sigma, \Omega, \Sigma' \vdash \Gamma \leq \text{ctx} \Gamma\] By Lemma 11 (i)

\[\Sigma, \Omega, \Sigma' \vdash \Gamma_0^+ \leq \text{ctx} \Gamma_0^+\] By above \((\Gamma_0^+ \vdash B^+) \leq \text{tk} (\Gamma_0 \vdash B)\)

\[\Sigma, \Omega, \Sigma' \vdash (\Gamma, \Gamma_0^+) \leq \text{ctx} (\Gamma, \Gamma_0^+)\] By Lemma 11 (iii)

\[\Sigma, \Omega, \Sigma'; \Gamma; p : A \vdash (\{p_1 \Rightarrow e_1\} \ ms : D)\] By i.h.

Then:

\[\Sigma, \Omega, \Sigma'; \Gamma_0; \{p \cap \neg p_1\} : A \vdash \text{ms} : D\]

Subderivation

\[\Sigma, \Omega, \Sigma'; \Gamma_0; \{p \cap \neg p_1\} : A \vdash \text{ms} : D\]

By i.h.

\[\Sigma, \Omega, \Sigma'; \Gamma_0^+; p : A \vdash \left((p_1 \Rightarrow e_1) \ \text{ms} : D\right)\] By TypeMs

Lemma 14 (Value substitution). Suppose \(\Sigma; \Gamma_1, \Gamma_R \vdash v : A\).

(i) If \(\Sigma; \Gamma_1, x : A, \Gamma_R \vdash e : B\) then \(\Sigma; \Gamma_1, \Gamma_R \vdash [v/x]e : B\).

(ii) If \(\Sigma; \Gamma_1, x : A, \Gamma_R, p : B \vdash \text{ms} : D\) then \(\Sigma; \Gamma_1, \Gamma_R, p : B \vdash [v/x]\text{ms} : D\).

Proof. In each part, by induction on the derivation specific to that part.

For part (i), in the \Var case, we have \(\Sigma; \Gamma_1, \Gamma_R \vdash v : A\), and \(v = [v/x]x\). In all other cases, use the i.h. on each premise, then apply the same rule. (For DataE use part (ii).)

For part (ii), we have two cases:

- **Case TypeMs** The intersect function does not depend on the typing context, so we get the same set of tracks. Apply the i.h. (i) to each typing subderivation and the i.h. (ii) to the last subderivation.

- **Case TypeMSEmpty** This rule does not depend on the typing context at all, so we just apply it.

Lemma 15 (Multiple substitution).

If \(\Sigma; \Gamma \vdash \theta : \Gamma'\) then \(\Sigma; \Gamma \vdash [\theta]e : B\).

Proof. By induction on the derivation of \(\Sigma; \Gamma \vdash \theta : \Gamma'\). In the SubstEmpty case, apply the equality \([\theta]e = [\theta]e = e\).

In the SubstVar case, use the i.h. and Lemma 14 (Value substitution).

B.4 Value inversion

A value inversion (or canonical forms) lemma holds:

**Lemma 16** (Inversion). Suppose \(\Sigma; \vdash v : B\).

(1) If \(\Sigma \vdash B \leq s\) then there exist \(c\) and \(v'\) such that \(v = c(v')\)

and \((c : A \rightarrow t) \in \Sigma\) and \(\Sigma \vdash t \leq s\) and \(\Sigma; \vdash v' : A\).

(2) If \(\Sigma \vdash B \leq (A_1 \rightarrow A_2)\)

then \(v = \lambda x. e\) and \(\Sigma; \vdash x : B_1 \vdash e : A_2\) where \(\Sigma \vdash A_1 \leq B_1\).

(3) If \(\Sigma \vdash B \leq 1\) then \(v = \langle\rangle\).
(4) If \( \Sigma \vdash B \leq (A_1 \ast A_2) \) then \( v = (v_1, v_2) \) where \( \Sigma ; \vdash v_1 : A_1 \) and \( \Sigma ; \vdash v_2 : A_2 \).

Proof. By induction on the given derivation.

- **Part (1):**
  We have \( \Sigma \vdash B \leq s \). By inversion on subtyping, \( B \) has the form \( B^* ::= t \mid B_1 \land B_2 \). Thus, the only possible cases are Sub, \( \land \) and DataI. \( \text{Declare} \) is impossible because a \textsc{declare} is not a value.
  
  - Case  \( \Sigma_1 ; \vdash v : B' \quad \Sigma \vdash B' \leq B \)  
    \[ \Sigma_1 ; \vdash v : B \quad \text{Subderivation} \]
    \[ \Sigma_1 ; \vdash v : B' \quad \text{Subderivation} \]
    \[ \Sigma \vdash B \leq s \quad \text{Given} \]
    \[ \Sigma \vdash B' \leq s \quad \text{By Lemma 7 (Transitivity)} \]
    The result follows by i.h.
  
  - Case  \( \Sigma_1 ; \vdash v : B_1 \quad \Sigma_1 ; \vdash v : B_2 \)
    \[ \Sigma_1 ; \vdash v : (B_1 \land B_2) \quad \land \]
    \[ \Sigma \vdash (B_1 \land B_2) \leq s \quad \text{Given} \]
    \[ \Sigma \vdash B_k \leq s \quad \text{By inversion} (\leq \land L_k) \]
    \[ \Sigma_1 ; \vdash v : B_k \quad \text{Subderivation} \]
    The result follows by i.h.
  
  - Case  \( \Sigma_1 ; \vdash c : A \rightarrow s_0 \quad \Sigma_1 ; \vdash v' : A \)
    \[ \Sigma_1 ; \vdash c(v') : s_0 \quad \text{DataI} \]
    \[ v = c(v') \quad \text{Above} \]
    \[ \Sigma_1 ; \vdash v' : A \quad \text{Subderivation} \]
    \[ \Sigma \vdash c : A \rightarrow s_0 \quad \text{Subderivation} \]
    \[ (c : A \rightarrow t) \in \Sigma \quad \text{By inversion} (\text{ConArr}) \]
    \[ \Sigma \vdash t \leq s_0 \quad " \]
    \[ \Sigma \vdash s_0 \leq s \quad \text{Given} \]
    \[ \Sigma \vdash s_0 \leq s \quad \text{By inversion} (\leq \text{Data}) \]
    \[ \Sigma \vdash t \leq s \quad \text{By} \leq \text{Trans} \]

- **Part (2):**
  The only possible cases are Sub, \( \land \) and \( \to \).
  
  - Cases  Sub, \( \land \)  
    Similar to the respective cases for part (1).
  
  - Case  \( \Sigma, \Gamma \vdash e : B_2 \)
    \[ \Sigma \vdash \lambda x : B_1 \vdash e : B_2 \quad \to \]
    \[ \Sigma, \Gamma \vdash \lambda x, e : (B_1 \to B_2) \quad \to \]}
\[ \Sigma \vdash (B_1 \rightarrow B_2) \leq (A_1 \rightarrow A_2) \quad \text{Given} \]
\[ \Sigma \vdash A_1 \leq B_1 \quad \text{By inversion} \]
\[ \Sigma \vdash B_2 \leq A_2 \quad " \]
\[ \Sigma, \Gamma, x : B_1 \vdash e : B_2 \quad \text{Subderivation} \]
\[ \Sigma, \Gamma, x : B_1 \vdash e : A_2 \quad \text{By \text{Sub}} \]

- \textbf{Part (3):} Similar to part (2), but with \( \mathbb{I} \) instead of \( \mathbb{H} \).
- \textbf{Part (4):} Similar to part (2), but with \( \mathbb{I} \) instead of \( \mathbb{H} \).

\[ \square \]

\section*{B.5 Operational semantics lemmas}

\textbf{Lemma 17.} If \( e \) is a value then there exists no \( e' \) such that \( e \rightarrow e' \).

\textit{Proof.} By induction on \( e \).

\[ \square \]

\section*{B.6 Type preservation and progress}

\textbf{Lemma 18} (Pattern intersection).
\textit{If} \( p_1 \) match \( v \rightarrow \theta_1 \) \textit{and} \( p_2 \) match \( v \rightarrow \theta_2 \) \textit{then} \( p_1 \cap p_2 \) match \( v \rightarrow \theta \).

\textit{Proof.} By mutual induction on \( p_1 \) and \( p_2 \).

- \textbf{Case} \( p_1 = \emptyset \) or \( p_2 = \emptyset \): Impossible: \( \emptyset \) match \( v \rightarrow \ldots \) is not derivable.
- \textbf{Case} \( p_1 = \_ \) or \( p_2 = \_ \):
  Consider the \( p_1 = \_ \) case; the \( p_2 = \_ \) case is similar. It is given that \( p_2 \) match \( v \rightarrow \theta_2 \). We have \( p_1 \cap p_2 = \_ \cap p_2 = p_2 \), so \( p_1 \cap p_2 \) match \( v \rightarrow \theta \) (letting \( \theta = \theta_2 \)).
- \textbf{Case} \( p_1 = p_{11} \sqcup p_{12} \):
  \( p_{11} \sqcup p_{12} \) match \( v \rightarrow \theta_1 \) Given
  By inversion (\text{MatchOr}), either \( p_{11} \) or \( p_{12} \) is matched. Suppose the former; the latter is similar.
  \[
  \begin{align*}
  p_{11} \text{ match } v & \rightarrow \theta_1 \quad \text{By inversion MatchOr} \\
  p_{11} \cap p_2 \text{ match } v & \rightarrow \theta \quad \text{By i.h.} \\
  (p_{11} \cap p_2) \sqcup (p_{12} \cap p_2) \text{ match } v & \rightarrow \theta \quad \text{By MatchOr} \\
  (p_{11} \sqcup p_{12}) \cap p_2 & = (p_{11} \cap p_2) \sqcup (p_{12} \cap p_2) \quad \text{By def. of } \cap \\
  p_1 \cap p_2 \text{ match } v & \rightarrow \theta \quad \text{By above equality}
  \end{align*}
  \]
- \textbf{Case} \( p_2 = p_{21} \sqcup p_{22} \): Similar to the previous case.
- \textbf{Case} \( p_1 = c(p'_1) \):
  By inversion on \( c(p'_1) \) match \( v \rightarrow \theta_1 \), we have \( v = c(v') \) and \( p'_1 \) match \( v' \rightarrow \theta_1 \).
  We already dealt with the cases for \( p_2 \) being \( \emptyset, \_ \) or a \( \sqcup \). So by inversion on \( p_2 \) match \( c(v') \rightarrow \theta_2 \), we have \( p_2 = c(p'_2) \) and \( p'_2 \) match \( v' \rightarrow \theta_2 \).
  By the definition of \( \cap \), we have \( c(p'_1) \cap c(p'_2) = c(p'_1 \cap p'_2) \). By i.h., \( p'_1 \cap p'_2 \) match \( v' \rightarrow \theta' \).
  By MatchCon \( c(p'_1 \cap p'_2) \) match \( c(v') \rightarrow \theta' \).
Case \( p_1 = (p_{11}, p_{12}) \):
By inversion on \((p_{11}, p_{12})\) match \( v \rightarrow \theta_1 \), we have \( v = (v_{11}, v_{12}) \). By inversion on \( (v_{11}, v_{12}) \) match \( v \rightarrow \theta_2 \), we have \( v_{21} \rightarrow \theta_{21} \) and \( v_{22} \rightarrow \theta_{22} \) where \( \theta_2 = \theta_{21} \circ \theta_{22} \).

By the definition of \( \cap \), we have \((p_{11} \cap p_{21}, p_{21} \cap p_{22})\) match \( v \rightarrow \theta_1 \) and \( v \rightarrow \theta_2 \) where \( \theta_1 = \theta_{11} \circ \theta_{12} \).

By i.h., \( p_{11} \cap p_{21} \) match \( v_{11} \rightarrow \theta_1' \) and \( p_{21} \cap p_{22} \) match \( v_{22} \rightarrow \theta_2' \) where \( \theta_2' = \theta_{21} \circ \theta_{22} \).

By MatchCon, \((p_{11} \cap p_{21}, p_{21} \cap p_{22})\) match \( c(v') \rightarrow \theta_1' \circ \theta_2' \).

---

Lemma 19 (Excluded middle for matching).
If \( p \) match \( v \not\rightarrow \) then there exists no \( \theta \) s.t. \( p \) match \( v \rightarrow \theta \).

Proof. By induction on the given derivation of \( p \) match \( v \not\rightarrow \).

In all cases, at most one match rule could plausibly be applied, e.g. for \[NoMAs\] only \[MatchAs\] has a conclusion of the right form. In the \[NoMWild\] case, where \( p = \emptyset \), no match rule has a conclusion of the right form.

- Case \( \emptyset \) match \( v \not\rightarrow \) \[NoMWild\]
The match rules are directed by the syntax of the pattern, and there is no match rule with \( \emptyset \) in its conclusion.

- Case \( p_0 \) match \( v \not\rightarrow \) \[NoMAs\]
The only possibly applicable rule is \[MatchAs\] but by i.h., there exists no \( \theta \) such that \( p_0 \) match \( v \rightarrow \emptyset \).

- Case \( p_1 \) match \( v \not\rightarrow \) \( p_2 \) match \( v \not\rightarrow \) \[NoMOre\]
By i.h., neither \( p_1 \) match \( v \not\rightarrow \theta_1 \) nor \( p_2 \) match \( v \not\rightarrow \theta_2 \), so whether we choose \( k = 1 \) or \( k = 2 \), we can't apply \[MatchOr\].

- Cases \[NoMConHead\], \[NoMConInner\], \[NoMUnit\], \[NoMPairHead\], \[NoMPairInner\]
Straightforward, using the i.h. as needed.

---

Lemma 20 (Choice).
If \( \Sigma \) sig and \( U \vdash p : \tau \) and \( \Sigma ; : \vdash v : A \) and \( \Sigma \vdash A \subset \tau \) then either (1) \( p \) match \( v \rightarrow \emptyset \) or (2) \( p \) match \( v \not\rightarrow \) and \( \neg p \) match \( v \rightarrow \cdot \).

Proof. By structural induction on \( p \).

- Case \[p-Wild\] By \[MatchWild\] \( p \) match \( v \rightarrow \cdot \).
- Case p-As. We have \( p = (x \text{ as } p_0) \). By i.h., either \( p_0 \) match \( \nu \to \theta \), or \( p_0 \) match \( \nu \not\to \) and \( \lnot p_0 \) match \( \nu \to \cdot \).

  If \( p_0 \) match \( \nu \to \theta \), then by \( \text{MatchAs} \) \( x \text{ as } p_0 \) match \( \nu \to \theta \).

  Otherwise, by \( \text{NoMAs} \) \( x \text{ as } p_0 \) match \( \nu \not\to \). Above, we obtained \( \lnot p_0 \) match \( \nu \to \cdot \); by the definition of \( \lnot \), we have \( \lnot (x \text{ as } p_0) = \lnot p_0 \), so \( \lnot (x \text{ as } p_0) \) match \( \nu \to \cdot \).

- Case p-Empty. By \( \text{NoMWild} \) \( p \) match \( \nu \not\to \). By the definition of \( \lnot \), we have \( \lnot \emptyset = \cdot \).

  By \( \text{MatchWild} \) \( \_ \) match \( \nu \to \cdot \).

- Case \( \Sigma \vdash p_1 : \tau \quad \Sigma \vdash p_2 : \tau \quad p - \text{Or} \)

  If the i.h. gives \( p_1 \) match \( \nu \to \theta \), then by \( \text{MatchOr} \) \( p_1 \sqcup p_2 \) match \( \nu \to \theta' \).

  Otherwise, we have \( p_1 \) match \( \nu \not\to \) and \( \lnot p_1 \) match \( \nu \to \theta'' \).

    - If the i.h. on \( \Sigma \vdash p_2 : \tau \) gives \( p_2 \) match \( \nu \to \theta \), then by \( \text{MatchOr} \) \( p_1 \sqcup p_2 \) match \( \nu \to \theta' \).

    - Otherwise, we have \( p_2 \) match \( \nu \not\to \) and \( \lnot p_2 \) match \( \nu \to \theta'' \).

      By \( \text{NoMOr} \) \( p_1 \sqcup p_2 \) match \( \nu \not\to \).

      By the definition of \( \lnot \), we have \( \lnot (p_1 \sqcup p_2) = \lnot p_1 \cap \lnot p_2 \).

      By Lemma \( \text{15} \) (Pattern intersection), \( (\lnot p_1) \cap (\lnot p_2) \) match \( \nu \to \cdot \), which was to be shown.

- Case p-Unit.

  By Lemma \( \text{16} \) (Inversion) (3), \( v = \emptyset \).

  By \( \text{MatchUnit} \) \( \emptyset \) match \( \emptyset \) \( \to \cdot \).

- Case p-Pair

  We have \( p = (p_1, p_2) \) and \( \tau = \tau_1 \star \tau_2 \).

  By Lemma \( \text{16} \) (Inversion) (4), \( v = (v_1, v_2) \) and \( \Sigma; \vdash v_1 : B_1 \) and \( \Sigma; \vdash v_2 : B_2 \) where \( \Sigma \vdash B_1 \leq A_1 \) and \( \Sigma \vdash B_2 \leq A_2 \).

  If the i.h. gives (1), then we have \( p_1 \) match \( v_1 \to \theta_1 \).

    - By i.h. \( (p_2) \), we have either (1) \( p_2 \) match \( v_2 \to \theta_2 \) or (2) \( p_2 \) match \( v_2 \not\to \) and \( \lnot p_2 \) match \( v_2 \to \theta''_2 \).

      If (1), apply \( \text{MatchPair} \).

      If (2), then:

      By \( \text{NoMPairInner} \) \( (p_1, p_2) \) match \( v \not\to \).

      By \( \text{MatchWild} \) and \( \text{MatchPair} \) \( (\land \lnot p_2) \) match \( v \to \theta'_2 \).

      By \( \text{MatchOr} \) \( (\lnot p_1, \land \lnot \lnot p_2) \) match \( v \to \theta'_2 \).

      By the definition of \( \land \), this is \( (\lnot p_1, p_2) \) match \( (v_1, v_2) \to \theta'_2 \).

    Otherwise, we have \( p_1 \) match \( v_1 \not\to \) and \( \lnot p_1 \) match \( v_1 \to \theta'_1 \).

    - By rule \( \text{NoMPairInner} \) \( (p_1, p_2) \) match \( v \not\to \).

      By \( \text{MatchWild} \) and \( \text{MatchPair} \) \( (\lnot p_1, \land \lnot p_2) \) match \( v \to \theta'_1 \).

      By \( \text{MatchOr} \) \( (\lnot p_1, \land \lnot \lnot p_2) \) match \( (v_1, v_2) \to \theta'_1 \).

      By the definition of \( \land \), this is \( (\lnot p_1, p_2) \) match \( (v_1, v_2) \to \theta'_1 \).

- Case p-Cond.

  We have \( p = \sigma(p_0) \).

  By Lemma \( \text{16} \) (Inversion) (1), \( v = c_0(v_0) \) and \( \Sigma; \vdash v_0 : A_0 \) and \( (c_0 : A_0 \to s_0) \in \Sigma \) and \( \Sigma \vdash s_0 \leq s \).
If \( c \neq c_0 \), then:
\[
\text{match } v_0 \rightarrow \theta \quad \text{By } \text{NoMConHead}
\]
\[
\text{match } v_0 \rightarrow \cdot \quad \text{By } \text{MatchWild}
\]
\[
\text{match } c_0(v_0) \rightarrow \cdot \quad \text{By } \text{MatchCon}
\]
\[
-c(p_0) \sqcup \ldots \sqcup c_0(\_ \sqcup \ldots \text{match } c_0(v_0) \rightarrow \cdot \quad \text{By } \text{MatchOr}
\]
\[
-(c(p_0)) \text{match } c_0(v_0) \rightarrow \cdot \quad \text{By def. of } -
\]

If \( c = c_0 \), then:
1. \( U \vdash p_0 : \tau_0 \) By inversion on \( p \)-Con
2. \( \Sigma ; \vdash v_0 : A_0 \) Above
\( (c_0 : A_0 \rightarrow s_0) \in \Sigma \) Above
\( \Sigma \vdash c : A_0 \rightarrow s_0 \text{ contype} \) By inversion on \( \Sigma \) \text{ sig}
3. \( \Sigma \vdash A_0 \sqsubseteq \tau_0 \) By inversion on \( \text{ContypeArr} \)

By i.h. on (1, 2, 3), either:
- (1) \( p_0 \) match \( v_0 \rightarrow \theta \):
  \( c(p_0) \text{ match } c(v_0) \rightarrow \theta \) By \( \text{MatchCon} \)
- (2) \( p_0 \) match \( v_0 \) \( \not\rightarrow \) and \( -p_0 \) match \( v_0 \rightarrow \cdot \):
  \( c(p_0) \text{ match } c(v_0) \not\rightarrow \cdot \) By \( \text{NoMConInner} \)
  \( c(-p_0) \text{ match } c(v_0) \rightarrow \cdot \) By \( \text{MatchCon} \)
  \( c(-p_0) \sqcup c_1(\_ \sqcup \ldots \sqcup c_n(\_ \text{match } c(v_0) \rightarrow \cdot \) By \( \text{MatchOr} \)
  \( -c(p_0)) \text{match } c(v_0) \rightarrow \cdot \) By def. of \( -\)

**Theorem 5 (Intersect).** If \( \Sigma \text{ sig} \) and \( \Sigma ; \vdash v : A \) and \( \Sigma \vdash A \text{ type} \) and \( p \) match \( v \rightarrow \theta \) and \( \text{intersect}(\Sigma \vdash A; p) = B^* \) then there exists \( (\Gamma' \vdash B) \in B^* \) s.t. \( \Sigma ; \vdash v : B \) and \( \Sigma ; \vdash \theta : \Gamma' \) where \( \Sigma \vdash B \text{ type} \) and \( \Sigma \vdash B \leq A \).

**Proof.** By structural induction on \( p \).
Case-analyze the clause of the definition of the intersect function:
- Case: \( p = \_ \):
  \( \_ \text{match } v \rightarrow \theta \) Given
  \( \theta = \cdot \) By inversion \( \text{(MatchWild)} \)
  \( B^* = \{;\vdash A\} \) By def. of intersect
  \( \Sigma ; \vdash v : A \) Given
  \( \Sigma ; \vdash v : B \) \( B = A \)
  \( \cdot \vdash \cdot : \Gamma' \) By \( \text{SubstEmpty (} \Gamma' = \cdot \) \)
  \( \Sigma \vdash B \leq A \) By Lemma \( \delta (\text{Reflexivity}) \)
- Case: \( p = \emptyset \):
  We have \( \emptyset \text{ match } v \rightarrow \theta \), which is not derivable: this case is impossible.
– Case: $p = x$ as $p_0$:

\[
x \text{ as } p_0 \text{ match } v \rightarrow \theta \quad \text{Given}
\]
\[
p_0 \text{ match } v \rightarrow \theta \quad \text{By inversion [MatchAs]}
\]

$\Sigma; \vdash v : A$ \quad \text{Given}

\[
\text{intersect}(\Sigma \vdash A; x \text{ as } p_0) = \vec{B}^\ast \quad \text{Given}
\]

$\Gamma' = (\Gamma'_0, x : B)$ and $(\Gamma'_0 \vdash B) \in \text{intersect}(\Sigma \vdash A; p_0)$ for all $(\Gamma' \vdash B) \in \vec{B}^\ast$

By definition of intersect

\[
\Sigma; \vdash v : B \quad \text{By i.h.}
\]
\[
(\Gamma'_0 \vdash B) \in \vec{B}^\ast \quad "
\]
\[
\vdash \theta_0 : \Gamma'_0 \quad "
\]
\[
\Sigma \vdash B \leq A \quad "
\]
\[
\vdash (\theta_0, v/x) : (\Gamma'_0, x : B) \quad \text{By [SubstVar]}
\]

– Case: $p = p_1 \sqcup p_2$:

\[
\text{intersect}(\Sigma \vdash A; p_1 \sqcup p_2) = \vec{B}^\ast \quad \text{Given}
\]

\[
p_1 \sqcup p_2 \text{ match } v \rightarrow \theta \quad \text{Given}
\]
\[
p_1 \text{ match } v \rightarrow \theta \quad \text{By inversion [MatchOr] wlog}
\]

$(\Gamma' \vdash B) \in \text{intersect}(\Sigma \vdash A; p_1)$ \quad By i.h.

\[
\Sigma; \vdash v : B \quad "
\]
\[
\Sigma; \vdash \theta : \Gamma' \quad "
\]
\[
\Sigma; \vdash B \leq A \quad "
\]

\[
\text{intersect}(\Sigma \vdash A; p_1) \subseteq \vec{B}^\ast \quad \text{By def. of intersect}
\]

$(\Gamma' \vdash B) \in \vec{B}^\ast \quad \text{By a property of } \in$

– Case: $p = (p_1, p_2)$:

Throughout this case, interpret $k$ as universally quantified. For example, Lemma [10] [Inversion] (4) shows both $\Sigma; \vdash v_1 : A_1$ and $\Sigma; \vdash v_2 : A_2$. 

(p₁, p₂) match v → θ  
  Given
  v = (v₁, v₂)
  \theta = (\theta₁, \theta₂)
  p_k match v_k → \theta_k
  \Sigma;: \vdash (v₁, v₂): A₁ * A₂
  \Sigma;: \vdash v_k : A_k
  By Lemma 16 [Inversion] (4)

(Γ_k \vdash B_k) ∈ \text{intersect}(Σ \vdash A_k; p_k)  
  By i.h.
  \Sigma;: \vdash v_k : A_k
  \Sigma;: \vdash \theta_k : \Gamma_k
  \Sigma \vdash B_k \leq A_k
  \Sigma;: \vdash (v₁, v₂) : B₁ * B₂
  By apply
  \Sigma;: \vdash (\theta₁, \theta₂) : (Γ₁, Γ₂)
  By properties of substitution
  \Sigma \vdash (B₁ * B₂) \leq (A₁ * A₂)
  By apply

- Case: p = c(p₀):
  c(p₀) match v → θ  
  Given
  v = c(v₀)
  p₀ match v₀ → θ
  \Sigma;: \vdash c(v₀) : s
  \Sigma;: \vdash v₀ : A₀
  By Lemma 16 [Inversion] (1)
  (c : A₀ → s) ∈ \Sigma
  \Sigma \vdash s \leq s
  \Sigma \vdash \text{intersect}(Σ \vdash s; c(p₀)) = \vec{B}^\prime
  Given
  (Γ' \vdash B₀) ∈ \text{intersect}(Σ \vdash A₀; p₀)
  By def. of intersect
  \Sigma;: \vdash v₀ : B₀
  \Sigma;: \vdash \theta : \Gamma'
  \Sigma \vdash B₀ \text{ type}
  \Sigma \vdash B₀ \leq A₀
  By apply
  \Sigma;: \vdash \Gamma' \vdash v₀ : A₀
  By apply

  (Γ' \vdash s_c) ∈ \text{intersect}(Σ \vdash s; c(p₀))
  By def. of intersect
  \Sigma;: \vdash c(v₀) : s_c
  \Sigma;: \vdash c(v₀) : s
  By Sub
  \Sigma;: \vdash \theta : \Gamma'
  By DataI

\textbf{Lemma 21} (Match preservation).

If \Sigma sig and Σ \vdash A type and p match v → θ and Σ;:p : A \vdash ms : D
and ms \rightarrow e'
then Σ;: \vdash e' : D.

\textit{Proof.} By induction on the derivation of \cdots \vdash ms : D.

- Case \textbf{TypeMsEmpty.}
Given
\[ \Sigma; p : A \vdash \emptyset : D \]
By Theorem 5, there exists a track in \( \text{intersect}(\Sigma \vdash A; p) \) such that certain conditions hold. But \( \text{intersect}(\Sigma \vdash A; p) = \emptyset \), a contradiction. Thus, this case is impossible.

- Case for all \( (\Gamma' \vdash B) \)
\[ \Sigma \vdash A \sqsupset \tau \]
\[ U \vdash p_1 : \tau \]
\[ \Sigma; \Gamma' \vdash e_1 : D \]
\[ \Sigma; (p \cap \neg p_1) : A \vdash ms' : D \]
\( \text{TypeMs} \)
\[ \Sigma; p : A \vdash ((p_1 \Rightarrow e_1) \mid ms') : D \]
\( \text{ms} \)
\[ \Sigma \text{ sig} \]
\[ U \vdash p : \tau \]
\[ \Sigma; \vdash v : A \]
\[ \Sigma \vdash A \sqsubset \tau \]

If the derivation of \( ms \mapsto e' \) was concluded by \( \text{StepMatch} \) then \( e' = [\theta_1]e_1 \).
\[ \Sigma \text{ sig} \]
\[ \Sigma \vdash v : A \]
\[ \Sigma \vdash A \text{ type} \]
\[ p \text{ match } v \mapsto \emptyset \]
\[ (p \cap p_1) \text{ match } v \mapsto \theta_1 \]
\[ (\Gamma_1 \vdash B) \in \text{intersect}(\Sigma \vdash A; p \cap p_1) \]
\[ \Sigma; \vdash v : B \]
\[ \Sigma; \vdash \theta_1 : \Gamma_1 \]
\( \Sigma; ; \vdash e_1 : D \]
\( \Sigma; ; \vdash [\theta_1]e_1 : D \]
\( \Sigma; ; \vdash \theta_1 : \Gamma_1 \)
\[ \Sigma; ; \vdash e' : D \]

Otherwise, the derivation was concluded by \( \text{StepElse} \) where \( p_1 \text{ match } v \neq \emptyset \).
\[ ms' \mapsto e' \]
\[ \Sigma; ; (p \cap \neg p_1) : A \vdash ms' : D \]
\[ p \text{ match } v \mapsto \emptyset \]
\[ \neg p_1 \text{ match } v \mapsto \emptyset \]

The preservation result allows for a longer signature, to model entering the scope of a \textbf{declare} expression or the arms of a \textbf{match}. We implicitly assume that throughout the given typing derivation, all types are well-formed under the local signature: whenever we have a subderivation of \( \Sigma; \Gamma \vdash e' : B \), it is the case that \( \Sigma \vdash B \text{ type} \).

\[ \square \]
Theorem 6 (Preservation). If $\Sigma \vdash_\text{sig} e : A$ and $e \mapsto e'$ then there exists $\Sigma'$ such that $\Sigma, \Sigma' \vdash_\text{sig} e' : A$ where $(\Sigma, \Sigma') \text{ sig}.$

Proof. By induction on the derivation of $\Sigma; \cdot \vdash_\text{sig} e : A$.

The rules $\text{Var}, \text{1I}, \text{→I}$, and $\text{∧I}$ can only type values, which cannot step (Lemma 17), contradicting $e \mapsto e'$: these cases are impossible.

In most cases, the conditions about $\Sigma'$ follow directly from the i.h.

- Case $\text{→E}$ We have $e_1 \mapsto e'_1$ or $e_2 \mapsto e'_2$. Apply the i.h. to the appropriate subderivation, use the weakening lemma on the other subderivation, and apply $\text{→E}$.

- Case $\Sigma; \cdot \vdash_\text{sig} e : B \Sigma \vdash B \leq A$ Subderivation
  $\Sigma; \cdot \vdash e : A$ Subderivation
  $\Sigma, \Sigma' \vdash e' : B$ By i.h.
  $\Sigma \vdash B \leq A$ Subderivation
  $\Sigma, \Sigma' \vdash B \leq A$ By Lemma 8 (Weakening (lowest level)) (iii)
  $\Sigma, \Sigma' \vdash e' : A$ By Sub

- Case $\Sigma \vdash c : B \rightarrow s$ $\Sigma; \cdot \vdash e_0 : B$ DataI
  $\Sigma; \cdot \vdash c(e_0) : s$ Subderivation

By inversion on $c(e_0) \mapsto e'$ we have $e' = c(e'_0)$ and $e_0 \mapsto e'_0$.

- Case $\Sigma; \cdot \vdash e_0 : B$ Subderivation
  $\Sigma, \Sigma' \vdash e'_0 : B$ By i.h.
  $\Sigma \vdash c : B \rightarrow s$ Subderivation
  $\Sigma, \Sigma' \vdash c : B \rightarrow s$ By Lemma 8 (Weakening (lowest level)) (iv)
  $\Sigma, \Sigma' \vdash c(e'_0) : s$ Subderivation

- Case $\Sigma; \cdot \vdash e : B$ $\Sigma; \cdot \vdash_\text{ms} : A$ DataE
\[ \Sigma \text{ sig} \] Given
\[ \Sigma \vdash B \text{ type} \] By implicit assumption
\[ \text{\_match } v \rightarrow \_ \] By \text{MatchWild}
\[ \Sigma; \vdash B \vdash \text{ms : A} \] Subderivation
\[ \Sigma; \vdash e' : A \] By Lemma 21 \text{(Match preservation)}
\[ \text{case e of ms } \rightarrow e' \] By \text{StepCase}

- **Case**: \((\Sigma, \Sigma') \text{ sig } \Sigma \vdash A \text{ type} \)  \(\Sigma, \Sigma' ; \vdash e_0 : A\]

\[ \Sigma; ; (\text{declare } \Sigma' \text{ in } e_0) : A \] Declare

We have \((\text{declare } \Sigma' \text{ in } e_0) \mapsto e'\). By inversion \text{StepDeclare}, \(e' = e_0\).

\[ \langle \Sigma, \Sigma' \rangle \text{ sig} \] Premise
\[ \Sigma, \Sigma' ; \vdash e_0 : A \] Subderivation
\[ \Sigma, \Sigma' ; \vdash e_0 : A \] \(e' = e_0\)

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}

\[ \Sigma \text{ sig} \] Given
\[ \U \vdash p : \tau \] Subderivation
\[ \Sigma; \vdash v : A \] Given
\[ \Sigma \vdash A \sqsubseteq \tau \] Subderivation

By Lemma 20 \text{(Choice)}, either (1) \(p_1 \text{ match } v \rightarrow \theta_1\), or (2) \(p_1 \text{ match } v \not\rightarrow \) and \(\neg p_1 \text{ match } v \rightarrow \).

For case (1), apply \text{StepMatch}

For case (2), show \(p \cap \neg p_1 \text{ match } v \rightarrow \theta'\) as in the proof of Lemma 21 \text{(Match preservation)}, apply the i.h. to \(ms'\), then apply \text{StepElse}
– **Cases**  
  
  \[ \begin{array}{l}
  \text{I} \rightarrow \text{I} \rightarrow \text{I} : \text{The rule requires that } e \text{ is a value.}
  \\
  \text{E} : \text{We have } e = e_1 e_2. \text{ By i.h. on the subderivation typing } e_1, \text{ either } e_1 \text{ steps or } e_1 \text{ is a value:}
  \\
  \quad \bullet \text{ If } e_1 \text{ steps, the result follows by } \text{StepContext.}
  \\
  \quad \bullet \text{ If } e_1 \text{ is a value, then by i.h. on the subderivation typing } e_2, \text{ either } e_2 \text{ steps or } e_2 \text{ is a value.}
  \\
  \text{In the former case, the result follows by } \text{StepContext.}
  \\
  \text{In the latter case:}
  \\
  \quad \Sigma; \cdot \vdash e_1 : A' \rightarrow A \quad \text{Subderivation}
  \\
  \quad e_1 \text{ is a value} \quad \text{Above}
  \\
  \quad e_1 = (\lambda x. e_0) \quad \text{By Lemma 16 (Inversion) (2)}
  \\
  \quad (\lambda x. e_0) e_2 \leftrightarrow [e_2/x]e_0 \quad \text{By StepBeta}
  \\
  \end{array} \]

– **Case**  
  
  \[ \begin{array}{l}
  \ast \text{I} : \text{We have } e = (e_1, e_2).
  \\
  \text{By the i.h. on the subderivation typing } e_1, \text{ either } e_1 \text{ steps or } e_1 \text{ is a value.}
  \\
  \text{In the former case, the result follows by } \text{StepContext.}
  \\
  \text{In the latter case, use the i.h. on the subderivation typing } e_2. \text{ If } e_2 \text{ steps, apply } \text{StepPairR. Otherwise, } (e_1, e_2) \text{ is a value.}
  \\
  \end{array} \]

– **Case**  
  
  \[ \begin{array}{l}
  \text{Data} \text{I} : \text{We have } e = c(e_0).
  \\
  \text{By the i.h. on the subderivation typing } e_0, \text{ either } e_0 \text{ steps or } e_0 \text{ is a value.}
  \\
  \text{In the former case, the result follows by } \text{StepContext.}
  \\
  \text{In the latter case, } c(e_0) \text{ is a value.}
  \\
  \end{array} \]

– **Case**  
  
  \[ \begin{array}{l}
  \text{DataE} : \text{We have } e = \text{case } e_0 \text{ of } ms.
  \\
  \text{By the i.h. on the subderivation typing } e_0, \text{ either } e_0 \text{ steps or } e_0 \text{ is a value.}
  \\
  \text{In the former case, the result follows by } \text{StepContext.}
  \\
  \text{In the latter case:}
  \\
  \quad \Sigma \text{ sig} \quad \text{Given}
  \\
  \quad \_ \text{match } e_0 \rightarrow \text{.} \quad \text{By } \text{MatchWild}
  \\
  \quad \Sigma; \cdot : B \vdash ms : A \quad \text{Subderivation}
  \\
  \quad \text{case } e_0 \text{ of } ms \rightarrow e' \quad \text{By Lemma 22 (Match progress)}
  \end{array} \]

– **Case**  
  
  \[ \begin{array}{l}
  \text{Declare} \text{ } \text{The result follows by } \text{StepDeclare.}
  \end{array} \]

B.7 Bidirectional typing: soundness and completeness

**Theorem 8** (Bidirectional soundness). If \( \Gamma \vdash e \Leftarrow A \) or \( \Gamma \vdash e \Rightarrow A \) then \( \Gamma \vdash \lfloor e \rfloor : A \) where \( \lfloor e \rfloor \) is \( e \) with all annotations erased.

**Proof.** By induction on the given derivation.

In the \text{SynAnno} case, \( e = (e_0 : A_s) \). Apply the i.h. to get \( \Gamma \vdash \lfloor e_0 \rfloor : A \). Since \(|\lfloor e_0 : A_s \rfloor| = |e_0| \), we have \( \Gamma \vdash \lfloor (e_0 : A_s) \rfloor : A \), which was to be shown.

In all other cases, apply the i.h. to each subderivation and apply the type assignment rule corresponding to the bidirectional rule (\text{Var} for \text{SynVar}, \text{Sub} for \text{ChkSub}, and so on).

**Theorem 9** (Annotatability). If \( \Gamma \vdash e : A \) then:
(1) There exists $e_{\phi_1}$ such that $|e_{\phi_1}| = e$ and $\Gamma \vdash e_{\phi_1} \leftarrow A$.
(2) There exists $e_{\phi_2}$ such that $|e_{\phi_2}| = e$ and $\Gamma \vdash e_{\phi_2} \Rightarrow A$.

Proof. By induction on the given derivation.

For most cases: Use the i.h. on each subderivation and apply the corresponding bidirectional rule, using part (1) of the i.h. for checking premises and part (2) for synthesizing premises. If the conclusion of the corresponding bidirectional rule is a checking judgment, then part (1) has been shown; part (2) follows by adding an annotation and using $\text{SynAnno}$. Otherwise, the conclusion of the corresponding rule is a synthesis judgment; part (2) has been shown, and applying $\text{ChkSub}$ gives part (1).

For the $\land I$ case of part (1), we have $A = (A_1 \land A_2)$. The i.h. on the first subderivation yields $\Gamma \vdash e_{\phi_1} \leftarrow A_1$, and on the second subderivation it yields $\Gamma \vdash e_{\phi_2} \leftarrow A_2$. Let $e_{\phi_1}$ be $e$ with all annotations from $e_{\phi_1}$ and $e_{\phi_2}$. If $e_{\phi_1}$ and $e_{\phi_2}$ have different annotations on the same subterm, then $e_{\phi_1}$ has their union; for example, if $e_{\phi_1} = (e_0 : B_1)$ and $e_{\phi_2} = (e_0 : (B_2, B_3))$ then let $e_{\phi_1} = (e_0 : (B_1, B_2, B_3))$. Applying $\text{Chk} \land I$ gives $\Gamma \vdash e_{\phi_1} \leftarrow (A_1 \land A_2)$, which was to be shown.

For the $\land I$ case of part (2), follow part (1), then add an annotation, yielding $\Gamma \vdash (e_{\phi_1} : (A_1 \land A_2)) \Rightarrow (A_1 \land A_2)$.

B.8 Bidirectional typing: decidability

Lemma 23 (Decidability). Given instantiations of the meta-variables, the following judgments are decidable:

(1) the refinement judgment $\Sigma \vdash A < \tau$
(2) the well-formedness judgments $\Sigma \vdash A : \text{type}$ and $\Sigma \vdash c : \text{contype}$
(3) the subsorting judgment $\Sigma \vdash s_1 \preceq s_2$
(4) the constructor typing judgment $\Sigma \vdash c : C$
(5) the subtyping judgment $\Sigma \vdash A \leq B$
(6) the safe extension judgments $\Sigma ; S(K) \vdash c : C \text{ safe at } t$ and $\Sigma ; S(K) \vdash K_{\text{elem}} \text{ safe}$
(7) the signature well-formedness judgment $\Sigma ; \text{sig}$
(8) the pattern type judgment $U \vdash p : \tau$

Moreover, the $\text{intersect}$ function is computable.

Proof. For (1) and (2), the type gets smaller in every nontrivial premise.

For subsorting (3), construct the transitive closure.
The single rule for (4) depends only on (3).
For subtyping (5), at least one type gets smaller in each premise.
The rules for (6) are not genuinely inductive, using only previous judgments.
For (7), the signature gets smaller in the first premise.
For (8), the pattern gets smaller in each premise.
In the definition of $\text{intersect}$, the pattern gets smaller in each recursion.
Theorem 10 (Decidability).
Given a signature $\Sigma$, context $\Gamma$ and expression $e$, the set of $A$ such that $\Sigma;\Gamma \vdash e \Rightarrow A$ is decidable; and, given also a type $B$, the judgment $\Sigma;\Gamma \vdash e \Leftarrow B$ is decidable.

Moreover, given a signature $\Sigma$, context $\Gamma$, pattern $p$, type $A$, matches $ms$ and type $B$, the judgment $\Sigma;\Gamma;_:-A \vdash ms \Leftarrow B$ is decidable.

Proof. The auxiliary judgments, such as subtyping ($\text{ChkSub}$) and well-formedness ($\text{ChkDeclare}$), are decidable by Lemma 23 (Decidability).

In each premise of each bidirectional typing rule, either

1. the expression gets smaller ($\text{SynAnno}$, $\text{Chk} \rightarrow \text{I}$, $\text{Syn} \rightarrow \text{E}$, $\text{Chk}\ast \text{I}$, $\text{ChkDataI}$, $\text{ChkDataE}$, $\text{ChkDeclare}$), or
2. the expression is the same; then, either
   (a) the conclusion is checking, the premise is checking, and the type gets smaller ($\text{Chk}\land \text{I}$), or
   (b) the conclusion is checking, and the premise is synthesizing ($\text{ChkSub}$), or
   (c) we are using $\text{Syn}\land \text{E}$.

That is, we can order the problems lexicographically, considering the expression first; then, we consider the synthesis problem smaller than the checking problem. For the rules $\text{Syn}\land \text{E}$ and $\text{Syn}\land \text{E}$ observe that, assuming the premise has been derived, each rule enumerates one part of the intersection; since type expressions are finite, only finitely many types can be so enumerated. \qed