Extensible Datasort Refinements

Jana Dunfield(✉)

University of British Columbia, Vancouver, Canada
jd169@queensu.ca

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:

```
case elems of head :: tail => head
```

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

Datasort refinements eliminate this problem: a datasort can express, within the static type system, that `elems` is not empty; therefore, the above case expression will never raise `Match`. Datasorts can also express less shallow properties. For example, the definition in Fig. 1 encodes conjunctive normal form—a formula that consists of (possibly nested) And*s* of clauses, where a clause consists of (possibly nested) 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

H. Yang (Ed.): ESOP 2017, LNCS 10201, pp. 476–503, 2017.
DOI: 10.1007/978-3-662-54434-1_18

$$
\begin{array}{rcl}
\mathsf{pos\text{-}literal} & \preceq & \mathsf{literal}, \quad \mathsf{literal} \preceq \mathsf{clause}, \quad \mathsf{clause} \preceq \mathsf{cnf}, \\
\mathsf{Var} & : & \mathsf{symbol} \to \mathsf{pos\text{-}literal}, \\
\mathsf{Not} & : & \mathsf{pos\text{-}literal} \to \mathsf{literal}, \\
\mathsf{Or} & : & (\mathsf{clause} * \mathsf{clause}) \to \mathsf{clause}, \\
\mathsf{And} & : & (\mathsf{cnf} * \mathsf{cnf}) \to \mathsf{cnf}
\end{array}
$$

Fig. 1. Datasorts for conjunctive normal form

values of type clause would only need branches for Or, Not and Var; the And branch could be omitted, since And does not produce a 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 (Freeman and Pfenning 1991; Freeman 1994) extended ML, using abstract interpretation (Cousot and Cousot 1977) 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 (Freeman and Pfenning 1991, p. 271).

Thus, the second generation of work on datasort refinements (Davies and Pfenning 2000; Davies 2005) 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 (Dunfield and

Pfenning 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.

$$\begin{array}{rcl} \mathsf{even} \preceq \mathsf{bits}, \quad \mathsf{odd} \preceq \mathsf{bits}, & & \\ \mathsf{Empty} & : & \mathsf{even}, \\ \mathsf{One} & : & (\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even}), \\ \mathsf{Zero} & : & (\mathsf{even} \to \mathsf{even}) \wedge (\mathsf{odd} \to \mathsf{odd}) \end{array}$$

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

$$\mathsf{b} : \mathsf{odd} \vdash \mathsf{One}(\mathsf{b}) : \mathsf{even}$$

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[1], 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.

$$\begin{aligned} \mathsf{even} &= \mathsf{Empty} \mid \mathsf{Zero}(\mathsf{even}) \mid \mathsf{One}(\mathsf{odd}) \\ \mathsf{odd} &= \mathsf{Zero}(\mathsf{odd}) \mid \mathsf{One}(\mathsf{even}) \end{aligned}$$

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 Tainted and Untainted:

$$\begin{aligned} \mathsf{Z} &: \mathsf{nat}, \qquad \mathsf{S} : \mathsf{nat} \to \mathsf{nat}, \\ \mathsf{tainted} &\preceq \mathsf{nat}, \quad \mathsf{untainted} \preceq \mathsf{nat}, \\ \mathsf{Z} &: \mathsf{tainted}, \quad \mathsf{S} : \mathsf{tainted} \to \mathsf{tainted}, \\ \mathsf{Z} &: \mathsf{untainted}, \quad \mathsf{S} : \mathsf{untainted} \to \mathsf{untainted} \end{aligned}$$

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

$$\mathsf{x} : \mathsf{tainted} \not\vdash \mathsf{S}(\mathsf{x}) : \mathsf{untainted}$$

Thus, the two sorts have different *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 (Kennedy 1996).

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 $\mathsf{untainted} \preceq \mathsf{tainted}$. Subsorting can be either *structural* (as the signatures generated from grammars) or *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.

Contributions. This paper makes the following contributions:

- A language and type system with *extensible signatures* for datasort refinements (Sect. 3). Refinements are extended by *blocks* that are checked to ensure that they do not weaken a sort's inversion principle, which would make typing unsound.
- A new formulation of typing (Sect. 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.
- Type (datasort) preservation and progress for the type assignment system, stated in Sect. 6 and proved in Appendix B, with respect to a standard call-by-value operational semantics (Sect. 5).
- A bidirectional type system (Sect. 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).

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 $2\mathsf{k}$ of a 0-based array of length $\mathtt{n}$, an indexed type system must check whether the proposition $2\mathsf{k} < \mathtt{n}$ is entailed in the theory of integers (under some set of assumptions, e.g. $0 \leq \mathsf{k} \leq \mathtt{n}/3$). The truth of $2\mathsf{k} < \mathtt{n}$ 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 Fig. 1, in which clause contains disjunctions of literals and cnf contains conjunctions of clauses, with a new sort for *conjunctions* of literals:

$$\begin{array}{ll} \text{[everything from Fig. 1]} & \mathsf{literal} \preceq \mathsf{conj\text{-}literal},\ \mathsf{conj\text{-}literal} \preceq \mathsf{cnf}, \\ & \mathsf{And} \ : \ (\mathsf{conj\text{-}literal} * \mathsf{conj\text{-}literal}) \to \mathsf{conj\text{-}literal} \end{array}$$

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 (Sect. 1) has a best, or principal, type: $\mathsf{bits} \to \mathsf{bits}$. 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 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*. Maybe the programmer only cares that the function preserves odd parity, and annotates it with $\mathsf{odd} \to \mathsf{odd}$; 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 Sect. 4.

$$
\begin{array}{lrcl}
\text{Term vars.} & x, y, \ldots & & \\
\text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \mid (e_1,\, e_2) \mid c(e) \mid \mathsf{case}\ e\ \mathsf{of}\ ms \\
& & & \mid \mathsf{declare}\ \Sigma\ \mathsf{in}\ e \quad \text{—signature extension (Fig. 4)} \\
\text{Matches} & ms & ::= & \cdot \mid \big((p \Rightarrow e)\ \mathbf{|}\ ms\big) \\
\text{Values} & v & ::= & x \mid \lambda x.\, e \mid (v_1,\, v_2) \mid c(v) \mid (v : A) \\
\text{Patterns} & p & ::= & \_ \mid \emptyset \mid c(p) \mid (p_1,\, p_2) \mid x\ \mathsf{as}\ p \mid p_1 \sqcup p_2
\end{array}
$$

Fig. 2. Expressions

3.1 Syntax

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

$$
\begin{array}{lrcl}
\text{Datasorts} & s, t, \ldots & & \\
\text{Types} & A, B, D & ::= & 1 \mid A \to B \mid A * B \mid s \mid A \wedge B \\
\text{Typing contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Fig. 3. Types and contexts

Types (Fig. 3), written A and B, include unit (1), function, and product types, along with datasorts s and t. The intersection type $A \wedge B$ 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 $(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{even})$: given any bitstring b, the repetition bb has even parity.

3.2 Unrefined Types and Signatures

Our unrefined types τ, in Fig. 4, are very simple: unit 1, functions $\tau_1 \to \tau_2$, products $\tau_1 * \tau_2$, and datatypes d. We assume that each datatype has a known set of constructors: for example, the bitstring type of Sect. 1 has constructors 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* $\mathcal{U}$ that gives datatype names (d) and (unrefined) constructor typings ($\mathsf{c} : \tau \to \mathsf{d}$). Since this signature is the same throughout a program, we elide it in most judgment forms.

The judgment $\Sigma \vdash A \sqsubset \tau$ says that A is a refinement of τ. Both the symbol $\sqsubset$ 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 $\to$ is covariant, not contravariant. Covariance is needed for functions whose domains are nontrivially refined, e.g. $\mathsf{odd} \to \cdots$, which is not a subtype of $\mathsf{bits} \to \cdots$ because $\mathsf{bits} \not\leq \mathsf{odd}$.

Rule $\sqsubset\wedge$ implements the usual refinement restriction: both parts of an intersection $A_1 \wedge A_2$ must refine the same unrefined type τ.

3.3 Signatures

Refinements are defined by *signatures* Σ (Fig. 4).

Unrefined datatype names		d		
Unrefined types		τ	$::=$	$1 \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid \mathsf{d}$
Unrefined signatures		$\mathcal{U}$	$::=$	$\cdot \mid \mathcal{U}, \mathsf{d} \mid \mathcal{U}, \mathsf{c} : \tau \to \mathsf{d}$
Constructor types	C	$::=$	$A \to s$	
Blocks	K	$::=$	$\cdot$	empty block
			$\mid K, s_1 \preceq s_2$	subsorting declaration
			$\mid K, \mathsf{c} : C$	constructor type decl.
Sort sets	S	$::=$	$(s_1 \sqsubset \mathsf{d}_1, \ldots, s_n \sqsubset \mathsf{d}_n)$	
Abbrev. sort sets	S	$::=$	$(s_1, \ldots, s_n)$	
Signatures	Σ, Ω	$::=$	$\cdot$	empty signature
			$\mid \Sigma, S\langle K \rangle$	datasort specification

$\boxed{\Sigma \vdash A \sqsubset \tau}$ Under signature Σ (and unrefined signature $\mathcal{U}$), type A is a refinement of unrefined type τ

$$\frac{}{\Sigma \vdash 1 \sqsubset 1}\sqsubset 1 \qquad \frac{\Sigma \vdash A_1 \sqsubset \tau_1 \qquad \Sigma \vdash A_2 \sqsubset \tau_2}{\Sigma \vdash (A_1 \to A_2) \sqsubset (\tau_1 \to \tau_2)}\sqsubset\to \qquad \frac{\Sigma \vdash A_1 \sqsubset \tau_1 \qquad \Sigma \vdash A_2 \sqsubset \tau_2}{\Sigma \vdash (A_1 * A_2) \sqsubset (\tau_1 * \tau_2)}\sqsubset *$$

$$\frac{(s \sqsubset \mathsf{d}) \in \Sigma}{\Sigma \vdash s \sqsubset \mathsf{d}}\sqsubset \mathsf{Data} \qquad \frac{\Sigma \vdash A_1 \sqsubset \tau \qquad \Sigma \vdash A_2 \sqsubset \tau}{\Sigma \vdash (A_1 \wedge A_2) \sqsubset \tau}\sqsubset\wedge$$

Fig. 4. Unrefined types and signatures, refined signatures, $\sqsubset$

$\boxed{\Sigma \vdash A\ \mathit{type}}$ Type A is well-formed

$$\frac{}{\Sigma \vdash 1\ \mathit{type}}\ \mathsf{WfType1} \qquad \frac{\Sigma \vdash A_1\ \mathit{type} \qquad \Sigma \vdash A_2\ \mathit{type}}{\Sigma \vdash A_1 * A_2\ \mathit{type}}\ \mathsf{WfType}*$$

$$\frac{\Sigma \vdash A_1\ \mathit{type} \quad \Sigma \vdash A_2\ \mathit{type}}{\Sigma \vdash A_1 \to A_2\ \mathit{type}}\ \mathsf{WfType}\to \qquad \frac{\Sigma \vdash A_1\ \mathit{type} \quad \Sigma \vdash A_1 \sqsubset \tau \quad \Sigma \vdash A_2\ \mathit{type} \quad \Sigma \vdash A_2 \sqsubset \tau}{\Sigma \vdash A_1 \wedge A_2\ \mathit{type}}\ \mathsf{WfType}\wedge$$

$$\frac{s \in S}{\Sigma, S\langle K\rangle, \Sigma' \vdash s\ \mathit{type}}\ \mathsf{WfTypeSort}$$

$\boxed{\Sigma \vdash c : C\ \mathit{contype}}$ Under (prefix) context Σ, the typing $c : C$ refines a typing in $\mathcal{U}$

$$\frac{\Sigma \vdash A\ \mathit{type} \qquad \Sigma \vdash s\ \mathit{type} \qquad (c : \tau \to d) \in \mathcal{U} \qquad \Sigma \vdash (A \to s) \sqsubset (\tau \to d)}{\Sigma \vdash c : A \to s\ \mathit{contype}}\ \mathsf{ContypeArr}$$

Fig. 5. Type well-formedness

As in past datasort systems, we separate signatures Σ from typing contexts Γ. Typing assumptions over term variables (x, y, etc.) in Γ can mention sorts declared in Σ, but the signature Σ cannot mention the term variables declared in Γ. Thus, our judgment for term typing will have the form $\Sigma; \Gamma \vdash e : A$, where the term e can include constructors declared in Σ and variables declared in Γ, and the type A can include sorts declared in Σ. Some judgments, like subsorting $\Sigma \vdash s \preceq t$ and subtyping $\Sigma \vdash A \leq B$, are independent of variable typing and don't include Γ 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 Γ, which is extended in subderivations that type λ-expressions and other binding forms. So the signature must be explicit in our judgment forms (Fig. 5).

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 $(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$. 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 $c : C_1 \wedge C_2 \wedge \cdots$. 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 $c : C_1 \wedge C_2$.

Structure of Signatures. A signature Σ is a sequence of *blocks* $\mathsf{S}\langle\mathsf{K}\rangle$ of declarations, where refinements declared in outer scopes in the program appear to the left of those declared in inner scopes.

Writing $(\mathsf{s}\sqsubset\mathsf{d})\langle\mathsf{K}\rangle$ declares s to be a sort refining some (unrefined) datatype d; however, we usually elide the datatype and write just $\mathsf{s}\langle\mathsf{K}\rangle$. The declarations K, called the *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 subsorting 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. $(\mathsf{s}_1\sqsubset\mathsf{d}_1, \mathsf{s}_2\sqsubset\mathsf{d}_2)\langle\mathsf{K}\rangle$, abbreviated as $(\mathsf{s}_1, \mathsf{s}_2)\langle\mathsf{K}\rangle$.

Writing $\mathsf{s}_1 \preceq \mathsf{s}_2$ says that s_1 is a subsort of s_2, and $\mathsf{c} : \mathsf{C}$ says that constructor c has type C, where C has the form $A \to \mathsf{s}$. A constructor c can be given more than one type: $\Sigma = (\mathsf{s}, \mathsf{s}_1, \mathsf{s}_2)\langle \mathsf{s}_1 \preceq \mathsf{s},\ \mathsf{s}_2 \preceq \mathsf{s},\ \mathsf{c} : \mathsf{s}_1{\to}\mathsf{s}_2,\ \mathsf{c} : \mathsf{s}_2{\to}\mathsf{s}_1\rangle$.

Adding inhabitants to a sort is only allowed within its block. Thus, the following signature is ill-formed, because $\mathsf{c}' : 1{\to}\mathsf{s}$ adds the value $\mathsf{c}'()$ to s, but $\mathsf{c}' : 1{\to}\mathsf{s}$ is not within s's block: $\mathsf{s}\langle\mathsf{c} : \mathsf{s}{\to}\mathsf{s}\rangle,\ \mathsf{t}\langle\mathsf{c}' : 1{\to}\mathsf{s}\rangle$. New sorts can be declared as subsorts and supersorts of each other, and of previously declared sorts: $\mathsf{s}\langle\mathsf{c}_1 : 1{\to}\mathsf{s}, \mathsf{c}_2 : 1{\to}\mathsf{s}\rangle,\ \mathsf{t}\langle\mathsf{t} \preceq \mathsf{s}, \mathsf{c}_2 : 1{\to}\mathsf{t}\rangle$.

However, a block cannot modify the subsorting relation between earlier sorts; "backpatching" $\mathsf{s}_1 \preceq \mathsf{s}_2$ into the first block, through a new intermediate sort t, is not permitted: The signature $\Sigma_* = (\mathsf{s}_1, \mathsf{s}_2)\langle \mathsf{c} : 1{\to}\mathsf{s}_1, \mathsf{c} : 1{\to}\mathsf{s}_2\rangle,\ \mathsf{t}\langle \mathsf{s}_1 \preceq \mathsf{t}, \mathsf{t} \preceq \mathsf{s}_2\rangle$ is not permitted even though it looks safe: sorts s_1 and s_2 have the same set of inhabitants—the singleton set $\{\mathsf{c}()\}$—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 $\mathsf{s}_1 \preceq \mathsf{s}_2$ 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 $\mathsf{S}\langle\mathsf{K}\rangle$ 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—DataI in Fig. 6—the signature Σ is separated from the ordinary context Γ (which contains typing assumptions of the form $x : A$). The typing of c is delegated to its first premise, $\Sigma \vdash c : A \to s$, so we need a way to derive this judgment. At the top of Fig. 6, we define a single rule 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 Σ, the logically weaker judgment that e has type B under more assumptions Σ, Σ'. (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 $\Sigma \vdash c : A \to s$, then $\Sigma, \Sigma' \vdash c : A \to s$. Under our formulation of the signature, this holds: If $c : A \to s$, then there exists $(c : A \to s') \in \Sigma$ such that $s' \preceq s$. Therefore, there exists $(c : A \to s') \in (\Sigma, \Sigma')$. Likewise, since $\Sigma \vdash s' \preceq s$, we also have $\Sigma, \Sigma' \vdash s' \preceq s$. One cannot use Σ' to withdraw a commitment made in Σ.[2]

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 $\mathsf{case}\ x :\ \mathsf{empty}\ \mathsf{of}\ \mathsf{Nil}() \Rightarrow ()$ which is exhaustive for the signature $\Sigma = (\mathsf{list}, \mathsf{empty})\langle \mathsf{empty} \preceq \mathsf{list},\ \mathsf{Nil} : 1 \to \mathsf{empty},\ \mathsf{Cons} : \mathsf{list} \to \mathsf{list}\rangle$ but not for

$$(\Sigma, \Sigma') \ = \ (\mathsf{list}, \mathsf{empty})\langle \mathsf{empty} \preceq \mathsf{list},\ \mathsf{Nil} : 1 \to \mathsf{empty},\ \mathsf{Cons} : \mathsf{list} \to \mathsf{list}\rangle, \\ \langle \mathsf{Cons} : \mathsf{list} \to \mathsf{empty}\rangle$$

Suppose we type-check the case expression under Σ, but then extend Σ to (Σ, Σ'). Evaluating the above case expression with $x = \mathsf{Cons}(\mathsf{Nil}())$ will "fall off the end". The inversion principle that "every empty has the form $\mathsf{Nil}()$" is valid under Σ, but with the additional type for Cons in Σ', that inversion principle becomes invalid under (Σ, Σ'). 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 Σ', the second constructor type for Cons had a smaller codomain than the first: the second type had empty, instead of list. Varying the codomain downward *can* be sound when the lower codomain is newly defined:

[2] Under the traditional formulation where each constructor has just one type in a signature, the relationship between the old signature Σ and the new signature would be slightly more complicated: the old signature might contain $c : C_1$, and the new signature $c : C_1 \wedge C_2$, and we would need to explicitly eliminate the intersection to expose the old type C_1. In our formulation, the new signature appends additional typings for c while keeping the typing $c : C_1$ intact.

$\Sigma, \Sigma'' = \Sigma, \mathsf{subempty}\langle \mathsf{subempty} \preceq \mathsf{empty}, \mathsf{Nil} : 1 \to \mathsf{subempty} \rangle$. Here, the inversion principle that every empty is Nil is still valid (along with the new inversion principle that every $\mathsf{subempty}$ is Nil). We only added information about a new sort $\mathsf{subempty}$, without changing the definition of list and 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 Σ with $\mathsf{Cons} : \mathsf{empty} \to \mathsf{list}$, which is the same as the type Σ has for 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 $\mathsf{Nil}()$, or (2) v has the form $\mathsf{Cons}(y)$ where y has type list". Reading off the new inversion principle for list from $\Sigma, \mathsf{Cons} : \mathsf{empty} \to \mathsf{list}$, we get "either (1) v has the form $\mathsf{Nil}()$, or (2) v has the form $\mathsf{Cons}(y)$ where y has type list, or (3) v has the form $\mathsf{Cons}(y)$ where y has type empty". Since empty is a subsort of list, part (3) implies part (2), and any case arm that checks under the assumption that $y : \mathsf{list}$ must also check under the assumption that $y : \mathsf{empty}$. 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 *co*domain gets *larger* is sound but pointless. For example, adding $\mathsf{Nil} : 1 \to \mathsf{list}$ to Σ has no effect, because (in the introduction form) we could use the old type $\mathsf{Nil} : 1 \to \mathsf{empty}$ with subsumption ($\mathsf{empty} \preceq \mathsf{list}$).

Moving the Domain Up. Making the domain of a constructor *larger* is unsound in general. To show this, we need a different starting signature Σ_2.

$$\Sigma_2 \;=\; (\mathsf{list}, \mathsf{empty}, \mathsf{nonempty})\langle \mathsf{empty} \preceq \mathsf{list}, \mathsf{nonempty} \preceq \mathsf{list}, \\ \mathsf{Nil} : 1 \to \mathsf{empty}, \mathsf{Cons} : \mathsf{empty} \to \mathsf{nonempty} \rangle$$

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 Σ_2 the following inversion principle for values v of sort $\mathsf{nonempty}$: "v has the form $\mathsf{Cons}(y)$ where y has type empty". If $x : \mathsf{nonempty}$ then $\mathsf{Case}\, x\, \mathsf{of}\, \mathsf{Cons}\,(\mathsf{Nil}()) \Rightarrow ()$ is exhaustive under Σ_2. Now, extend Σ_2: $\Sigma_2, \Sigma_2' = \Sigma_2, \langle \mathsf{Cons} : \mathsf{list} \to \mathsf{nonempty} \rangle$. For the signature Σ_2, Σ_2', the inversion principle for $\mathsf{nonempty}$ should be "(1) v has the form $\mathsf{Cons}(y)$ where y has type empty, or (2) v has the form $\mathsf{Cons}(y)$ 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 (Σ_2, Σ_2') allows. Concretely, the case expression above was exhaustive under Σ_2, but is not exhaustive under (Σ_2, Σ_2') because $\mathsf{Cons}(\mathsf{Cons}(\mathsf{Nil}()))$ 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 6 gives rules deriving the main typing judgment $\Sigma; \Gamma \vdash e : A$. The variable rule Var, the introduction ($\to$I) and elimination ($\to$E) rules for $\to$, and the introduction rules for the unit type (1I) and products ($*$I) are standard. Products can be eliminated via case e of $(x_1, x_2) \Rightarrow \cdots$, so they need no elimination rule.

Subsumption. A subsumption rule Sub incorporates subtyping, based on the subsort relation $\preceq$; see Sect. 3.7. Several of the subtyping rules express the same properties as elimination rules would; for example, anything of type $A_1 \wedge A_2$ has type A_1 and also type A_2. Consequently, we can omit these elimination rules without losing expressive power.

$\boxed{\Sigma \vdash c : C}$ Under signature Σ, constructor c has type C

$$\frac{(c : A \to s') \in \Sigma \qquad \Sigma \vdash s' \preceq s}{\Sigma \vdash c : A \to s}\ \text{ConArr}$$

$\boxed{\Sigma; \Gamma \vdash e : A}$ Under signature Σ and context Γ, expression e has type A

$$\frac{}{\Sigma; \Gamma, x : A, \Gamma' \vdash x : A}\ \text{Var} \qquad \frac{\Sigma; \Gamma \vdash e : A \qquad \Sigma \vdash A \leq B}{\Sigma; \Gamma \vdash e : B}\ \text{Sub}$$

$$\frac{\Sigma; \Gamma, x : A \vdash e : B}{\Sigma; \Gamma \vdash (\lambda x.\, e) : (A \to B)} \to\text{I} \qquad \frac{\Sigma; \Gamma \vdash e_1 : (A \to B) \qquad \Sigma; \Gamma \vdash e_2 : A}{\Sigma; \Gamma \vdash e_1\ e_2 : B} \to\text{E}$$

$$\frac{\Sigma; \Gamma \vdash v : A_1 \qquad \Sigma; \Gamma \vdash v : A_2}{\Sigma; \Gamma \vdash v : (A_1 \wedge A_2)} \wedge\text{I} \qquad \wedge\text{E}_k \text{ admissible via Sub} + \leq\wedge\text{L}_k$$

$$\frac{\Sigma; \Gamma \vdash e_1 : A_1 \qquad \Sigma; \Gamma \vdash e_2 : A_2}{\Sigma; \Gamma \vdash (e_1, e_2) : A_1 * A_2} *\text{I} \qquad \text{elimination via DataE with } (x_1, x_2) \Rightarrow \cdots$$

$$\frac{}{\Sigma; \Gamma \vdash () : 1}\ 1\text{I} \qquad \frac{\Sigma \vdash c : A \to s \qquad \Sigma; \Gamma \vdash e : A}{\Sigma; \Gamma \vdash c(e) : s}\ \text{DataI}$$

$$\frac{\Sigma; \Gamma \vdash e : A \qquad \Sigma; \_ : A \vdash ms : B}{\Sigma; \Gamma \vdash (\text{case } e \text{ of } ms) : B}\ \text{DataE} \qquad \frac{(\Sigma, \Sigma')\ sig \qquad \Sigma \vdash B\ type \qquad \Sigma, \Sigma'; \Gamma \vdash e : B}{\Sigma; \Gamma \vdash (\text{declare } \Sigma' \text{ in } e) : B}\ \text{Declare}$$

Fig. 6. Typing rules for constructors and expressions

Intersection. The introduction rule $\wedge$ I 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 Sub and subtyping (Sect. 3.7).

$$\frac{\Sigma;\Gamma \vdash e : A_1 \wedge A_2}{\Sigma;\Gamma \vdash e : A_1} \qquad \frac{\Sigma;\Gamma \vdash e : A_1 \wedge A_2}{\Sigma;\Gamma \vdash e : A_2}$$

Datasorts. Rule DataI introduces a datasort, according to a constructor type found in Σ (via the $\Sigma \vdash c : C$ judgment). Rule DataE examines an expression e of type A and checks matches ms under the assumption that the expression matches the wildcard pattern $\_$; see Sect. 4.

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

$\boxed{\Sigma \vdash A \leq B}$ A is a subtype of B

$$\frac{}{\Sigma \vdash 1 \leq 1}\leq 1 \qquad \frac{\Sigma \vdash A_1 \leq B_1 \qquad \Sigma \vdash A_2 \leq B_2}{\Sigma \vdash (A_1 * A_2) \leq (B_1 * B_2)}\leq * \qquad \frac{\Sigma \vdash s \preceq t}{\Sigma \vdash s \leq t}\leq \mathsf{Data}$$

$$\frac{\Sigma \vdash B_1 \leq A_1 \qquad \Sigma \vdash A_2 \leq B_2}{\Sigma \vdash (A_1 \to A_2) \leq (B_1 \to B_2)}\leq \to \qquad \frac{\Sigma \vdash A_k \leq B}{\Sigma \vdash (A_1 \wedge A_2) \leq B}\leq \wedge L_k \qquad \frac{\Sigma \vdash A \leq B_1 \qquad \Sigma \vdash A \leq B_2}{\Sigma \vdash A \leq (B_1 \wedge B_2)}\leq \wedge R$$

Fig. 7. Subtyping

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 $\leq\wedge\ L_1$ has a premise $A_1 \leq B$ and conclusion $(A_1 \wedge A_2) \leq B$, rather than just $(A_1 \wedge A_2) \leq A_1$ (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 $B \leq C$ (Fig. 7).

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 Fig. 8 from bottom to top, we start with well-formedness of signatures Σ *sig*. For each block $S\langle K\rangle$, rule SigBlock checks that the sorts S are not duplicates ($S \cap \mathsf{dom}(\Sigma) = \emptyset$), and then checks that (1) subsorting is conserved by K and (2) each element in K is *safe*.

$\boxed{\mathsf{dom}(\Sigma)}$ Domain (declared sorts) of Σ: $\mathsf{dom}(S_1\langle K_1\rangle, \ldots, S_n\langle K_n\rangle) = S_1 \cup \cdots \cup S_n$

$\boxed{\Sigma \vdash s_1 \preceq s_2}$ Sort s_1 is a subsort of s_2

$$\frac{}{\Sigma_L, S\langle \ldots, s_1 \preceq s_2, \ldots\rangle, \Sigma_R \vdash s_1 \preceq s_2}\ \preceq\mathsf{Assum}$$

$$\frac{s \in \mathsf{dom}(\Sigma)}{\Sigma \vdash s \preceq s}\ \preceq\mathsf{Refl} \qquad \frac{\Sigma \vdash s_1 \preceq s_2 \qquad \Sigma \vdash s_2 \preceq s_3}{\Sigma \vdash s_1 \preceq s_3}\ \preceq\mathsf{Trans}$$

$\boxed{\Sigma; S\langle K\rangle \vdash c : C \ \textit{safe at}\ t}$ C safely extends a type given by Σ for c

$$\frac{(c : A' \to s') \in \Sigma \qquad \begin{array}{c}\Sigma, S\langle K\rangle \vdash s' \preceq t \\ \Sigma, S\langle K\rangle \vdash s \preceq s'\end{array} \qquad \Sigma, S\langle K\rangle \vdash A \leq A'}{\Sigma; S\langle K\rangle \vdash c : A \to s \ \textit{safe at}\ t}\ \mathsf{SafeConAt}$$

$\boxed{\Sigma; S\langle K\rangle \vdash K_{elem}\ \textit{safe}}$ $K_{elem} \in K$ is safe for $\Sigma; S\langle K\rangle$

$$\frac{s_1, s_2 \in (\mathsf{dom}(\Sigma) \cup S)}{\Sigma; S\langle K\rangle \vdash s_1 \preceq s_2\ \textit{safe}}\ \mathsf{BlockSubsort}$$

$$\frac{s \in S \qquad \begin{array}{c}\Sigma, S\langle K\rangle \vdash c : A \to s\ \textit{contype} \\ \text{for all } t \in \mathsf{dom}(\Sigma) \\ \text{such that } \Sigma, S\langle K\rangle \vdash s \preceq t, \\ \Sigma; S\langle K\rangle \vdash c : A \to s\ \textit{safe at}\ t\end{array}}{\Sigma; S\langle K\rangle \vdash c : A \to s\ \textit{safe}}\ \mathsf{BlockCon}$$

$\boxed{\Sigma\ \textit{sig}}$ Signature Σ is well-formed

$$\frac{}{\cdot\ \textit{sig}}\ \mathsf{SigEmpty}$$

$$\frac{\begin{array}{ll}\Sigma\ \textit{sig} & \text{for all } t_1, t_2 \in \mathsf{dom}(\Sigma), \\ (S \cap \mathsf{dom}(\Sigma)) = \emptyset & (\Sigma \vdash t_1 \preceq t_2) \\ & \text{iff } (\Sigma, S\langle K\rangle \vdash t_1 \preceq t_2)\end{array} \qquad \begin{array}{l}\text{for all } K_{elem} \in K, \\ \Sigma; S\langle K\rangle \vdash K_{elem}\ \textit{safe}\end{array}}{\Sigma, S\langle K\rangle\ \textit{sig}}\ \mathsf{SigBlock}$$

Fig. 8. Signature well-formedness and subsorting

(1) Subsorting Preservation. The subsortings declared in K must not affect the subsort relation between sorts previously declared in Σ. 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 Σ_* in Sect. 3.3. This premise could also be written as $(\Sigma \vdash \preceq|_{\mathsf{dom}(\Sigma)}) = (\Sigma, S\langle K\rangle \vdash \preceq|_{\mathsf{dom}(\Sigma)})$, where $\preceq|_{\mathsf{dom}(\Sigma)}$ is the $\preceq$ relation restricted to sorts in $\mathsf{dom}(\Sigma)$.

(2a) Subsort Elements. Rule BlockSubsort checks that the subsorts are in scope.

(2b) Constructor Element Safety. Rule BlockCon's 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 $A \to s$ is well-formed. Finally, for all sorts t that were (1) previously declared (in $\mathsf{dom}(\Sigma)$) and (2) supersorts of the constructor's codomain ($s \preceq t$), the rule checks that the constructor is "safe at t".

The judgment $\Sigma; S\langle K\rangle \vdash c : A \to s$ *safe at* t says that adding the constructor typing $c : A \to s$ does not invalidate Σ's inversion principle for t. Rule SafeConAt checks that signature Σ already has a constructor typing $c : A' \to s'$, where $s' \preceq t$,

such that $A \leq A'$. Thus, any value $c(v)$ typed using $c : A \to s$ can already be typed using $c : A' \preceq s'$, which is a subsort of t, so the new constructor typing $c : A \to s$ 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 $\sqsubset$ (Fig. 4) is a closer analogy.

More subtly, SafeConAt also checks that $s \preceq s'$. Suppose we have the signature $\Sigma = (t, s_1, s_2)\langle s_1 \preceq t,\ s_2 \preceq t,\ c_1 : s_1,\ c_2 : s_2\rangle$ and extend it with $s\langle s \preceq t,\ c_1 : s\rangle$. (To focus on the issue at hand, we assume c_1 and c_2 take no arguments.) For the original signature Σ, 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 subsort of t.

If, instead, we start with $\Sigma' = (t, s_1, s_2)\langle c_1 : t,\ s_1 \preceq t,\ s_2 \preceq t,\ c_1 : s_1,\ c_2 : s_2\rangle$ then the inversion principle for t under Σ' 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 $t' \preceq t$, so the inversion principle (again, under Σ' before extension) is equivalent to "v has type t". Extending Σ' with $s\langle s \preceq t,\ c_1 : s\rangle$ would extend the inversion principle to say "if $v : t$ then v has type t, or v has type s", but since $s \preceq t$ the extended inversion principle is equivalent to that for t under Σ'.

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 : bits against the pattern y as One(Empty), we shouldn't settle on y : bits because the scrutinee x has type bits; we should descend into the pattern and observe that Empty : even and One : (even → odd), so y must have type odd.

Restricting the form of case expressions to a single layer of clearly disjoint patterns $c_1(x_1) \mid \ldots \mid c_n(x_n)$ 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 subsorting were always structural, as in Davies's system, we could exploit a handy equivalence between patterns and values: if the pattern is x as $c(p_0)$, let-bind x to $c(p_0)$ inside the case arm, letting rule DataI figure out the type of x. But with nominal subsorting, constructing a value is *not* equivalent; see Davies (2005, pp. 234–235) and Dunfield (2007b, pp. 112–113).

Our approach is to support the full syntax, including as-patterns. This approach was taken by Dunfield (2007b, Chap. 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.

Instead, we confine most of the complexity to a single mechanism: a function called 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 $\mathcal{U} \vdash p : \tau$ that says that pattern p matches values of unrefined type τ under the unrefined signature $\mathcal{U}$.

Rule DataE for case expressions (Fig. 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 DataE is $\_$.

Each arm, of the form $p_1 \Rightarrow e_1$, is checked by rule TypeMs (Fig. 10). The leftmost premises check that the type A corresponds to the pattern type τ. The middle "for all" checks e_1 under various assumptions produced by the intersect

$\boxed{\mathcal{U} \vdash p : \tau}$ Pattern p is suitable for values of unrefined type τ

$$\frac{}{\mathcal{U} \vdash \_ : \tau}\ \text{p-Wild} \qquad \frac{\mathcal{U} \vdash p : \tau}{\mathcal{U} \vdash (x \text{ as } p) : \tau}\ \text{p-As} \qquad \frac{}{\mathcal{U} \vdash \emptyset : \tau}\ \text{p-Empty} \qquad \frac{}{\mathcal{U} \vdash () : 1}\ \text{p-Unit}$$

$$\frac{\mathcal{U} \vdash p_1 : \tau \quad \mathcal{U} \vdash p_2 : \tau}{\mathcal{U} \vdash (p_1 \sqcup p_2) : \tau}\ \text{p-Or} \qquad \frac{\mathcal{U} \vdash p_1 : \tau_1 \quad \mathcal{U} \vdash p_2 : \tau_2}{\mathcal{U} \vdash (p_1, p_2) : \tau_1 * \tau_2}\ \text{p-Pair} \qquad \frac{\mathcal{U} \vdash c : (\tau \to d) \quad \mathcal{U} \vdash p : \tau}{\mathcal{U} \vdash c(p) : d}\ \text{p-Con}$$

Fig. 9. Pattern type rules

$\boxed{\Sigma; \Gamma; \mathsf{p} : \mathsf{A} \vdash \mathsf{ms} : \mathsf{D}}$ For a scrutinee of type A that matches residual pattern p, check each match in ms against D

$$\frac{\begin{array}{lll} & \text{for all } (\Gamma' \vdash \mathsf{B}) & \\ \Sigma \vdash \mathsf{A} \sqsubset \tau & \in \mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p} \cap \mathsf{p}_1): & \\ \mathcal{U} \vdash \mathsf{p}_1 : \tau & \Sigma; \Gamma, \Gamma' \vdash \mathsf{e}_1 : \mathsf{D} & \Sigma; \Gamma; (\mathsf{p} \cap \neg \mathsf{p}_1) : \mathsf{A} \vdash \mathsf{ms} : \mathsf{D} \end{array}}{\Sigma; \Gamma; \mathsf{p} : \mathsf{A} \vdash \big((\mathsf{p}_1 \Rightarrow \mathsf{e}_1) \mid \mathsf{ms}\big) : \mathsf{D}}\,\mathsf{TypeMs}$$

$$\frac{\mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p}) = \emptyset}{\Sigma; \Gamma; \mathsf{p} : \mathsf{A} \vdash \emptyset : \mathsf{D}}\,\mathsf{TypeMsEmpty}$$

Fig. 10. Match typing

function (Sect. 4.2) with respect to the pattern $\mathsf{p} \cap \mathsf{p}_1$, 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 $\mathsf{p} \cap \neg \mathsf{p}_1$. These operations are defined in the appendix (Fig. 13).

When typing reaches the end of the matches, $\mathsf{ms} = \emptyset$ in rule TypeMsEmpty, we check that the case expression is exhaustive by checking that intersect returns $\emptyset$. For case expressions that are syntactically exhaustive, such as a case expression 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 $\neg \mathsf{p}$ and pattern intersection $\mathsf{p}_1 \cap \mathsf{p}_2$ in the appendix (Fig. 13). For example, $\neg \_ = \emptyset$. No types appear in these definitions, but the complement of a constructor pattern $\mathsf{c}(\mathsf{p}_0)$ uses the (implicit) unrefined signature $\mathcal{U}$. Our definition of pattern complement never generates as-patterns, so we need not define intersection for as-patterns.

$\boxed{\mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p}) = \vec{\mathsf{B}^*}}$ Intersection of type A with pattern p where each B^* has the form $(\Gamma' \vdash \mathsf{B}')$

$$\begin{aligned}
\mathsf{intersect}(\Sigma \vdash \mathsf{A};\, _) &= \{(\cdot \vdash \mathsf{A})\} \\
\mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \emptyset) &= \emptyset \\
\mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{x}\ \mathsf{as}\ \mathsf{p}) &= \{(\Gamma', \mathsf{x} : \mathsf{B} \vdash \mathsf{B}) \mid (\Gamma' \vdash \mathsf{B}) \in \mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p})\} \\
\mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p}_1 \sqcup \mathsf{p}_2) &= \mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p}_1) \cup \mathsf{intersect}(\Sigma \vdash \mathsf{A};\, \mathsf{p}_2) \\
\mathsf{intersect}(\Sigma \vdash \mathsf{A}_1 * \mathsf{A}_2;\, (\mathsf{p}_1, \mathsf{p}_2)) &= \{(\Gamma_1, \Gamma_2 \vdash \mathsf{B}_1 * \mathsf{B}_2) \mid (\Gamma_1 \vdash \mathsf{B}_1) \in \mathsf{intersect}(\Sigma \vdash \mathsf{A}_1;\, \mathsf{p}_1) \\
&\qquad \text{and } (\Gamma_2 \vdash \mathsf{B}_2) \in \mathsf{intersect}(\Sigma \vdash \mathsf{A}_2;\, \mathsf{p}_2)\} \\
\mathsf{intersect}(\Sigma \vdash \mathsf{s};\, \mathsf{c}(\mathsf{p}_0)) &= \{(\Gamma' \vdash \mathsf{s}_\mathsf{c}) \mid (\mathsf{c} : \mathsf{A}_\mathsf{c} \to \mathsf{s}_\mathsf{c}) \in \Sigma \ \text{ and } \ \Sigma \vdash \mathsf{s}_\mathsf{c} \preceq \mathsf{s} \\
&\qquad \text{and } \ (\Gamma' \vdash \mathsf{B}) \in \mathsf{intersect}(\Sigma \vdash \mathsf{A}_\mathsf{c};\, \mathsf{p}_0)\}
\end{aligned}$$

Fig. 11. Intersection of a type with a pattern

4.2 The intersect function

We define a function intersect that builds the "intersection" of a type and a pattern. Given a signature Σ, 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 $(\Gamma' \vdash B)$ has a list of typings Γ' (giving the types of 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 Γ'.

Many of the clauses in the definition of intersect (see Fig. 10) are straightforward. The intersection of A with the wildcard _ 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 $p_1 \sqcup p_2$ is the union of two intersections (A with p_1, and A with p_2). The intersection of a product $A_1 * A_2$ 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 Σ that could have been used to create the given value: those of the form $(c : A_c \to s_c)$ where $s_c \preceq s$. For each such declaration, it calls intersect on A_c and p_0. For each resulting track $(\Gamma' \vdash B)$, it returns a track $(\Gamma' \vdash s_c)$.

Optimization. In practice, it may be necessary to optimize the result of intersect. If $\Sigma = (\mathsf{list}, \mathsf{empty})\langle \mathsf{empty} \preceq \mathsf{list}, \mathsf{Nil} : 1 \to \mathsf{empty}, \mathsf{Cons} : \mathsf{empty} \to \mathsf{list}, \mathsf{Cons} : \mathsf{list} \to \mathsf{list}\rangle$ then $\mathsf{intersect}(\Sigma \vdash \mathsf{Cons}(x \mathsf{as} \_); \mathsf{list})$ returns $\{(x : \mathsf{empty} \vdash \mathsf{list}),\ (x : \mathsf{list} \vdash \mathsf{list})\}$. Since any case arm that checks under $x : \mathsf{list}$ will check under $x : \mathsf{empty}$, there is no point in trying to check under $x : \mathsf{empty}$. Instead, we should check only under $x : \mathsf{list}$. 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 λ, so 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, $s_1 \wedge s_2$. 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, Chap. 4) appears to be a conservative extension of intersect: the earlier system supports a richer type system, but for the features in common, the information produced is similar to that of intersect. The earlier system was based on a judgment $\Sigma \vdash p \Leftarrow A \rhd (e \Leftarrow D)$. To clarify the connection to the present system, we adjust notation; for example, we make Σ explicit.

The meta-variables Σ, p, and A directly correspond to the arguments to 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 intersect, but within

$\Sigma \vdash p \Leftarrow A \rhd (e \Leftarrow B)$, we find subderivations of $B + \Gamma \vdash \textsc{forgettype} \rhd e \Leftarrow D$, where the set of pairs $\langle \Gamma, B \rangle$ indeed correspond to the result of $\mathsf{intersect}$.

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 $\mathsf{intersect}$. 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 $\wedge$ appears in the declarative system in Dunfield (2007b, Chap. 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 $\mathsf{intersect}$ 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 $e \mapsto e'$, which uses evaluation contexts ε. Stepping case expressions is modelled using a judgment $ms \mapsto_v e'$, which compares each pattern in ms against the value v being cased upon. This comparison is handled by the judgment $p \ \mathsf{match}\ v \longrightarrow \theta$, which says that θ is evidence that p matches v (that is, $[\theta]p = v$). The rules are in Fig. 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 Subsorting. Subtyping is reflexive and transitive (Lemmas (Lemmas 6–7). We define what it means for signature extension to preserve subsorting:

Definition 1 (Preserving subsorting). *Given* Σ_1 *and* Σ_2*, we say that* Σ_2 preserves subsorting *of* Σ_1 *iff for all sorts* $s, t \in \mathsf{dom}(\Sigma_1)$*, if* $\Sigma_1, \Sigma_2 \vdash s \preceq t$ *then* $\Sigma_1 \vdash s \preceq t$.

This definition allows new sorts in $\mathsf{dom}(\Sigma_2)$ to be subsorts or supersorts of the old sorts in $\mathsf{dom}(\Sigma_1)$, 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 subsorting.

Definition 2 (Non-adjacency). *Two signatures* Σ_1 *and* Σ_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* $s_1 \in \mathsf{dom}(\Sigma_1)$ *and* $s_2 \in \mathsf{dom}(\Sigma_2)$.

Theorem 1 (Non-adjacent preservation).
If Σ_2 *preserves subsorting of* Σ_1 *and* Σ_3 *preserves subsorting of* Σ_1 *and* Σ_2 *and* Σ_3 *are non-adjacent then* Σ_3 *preserves subsorting of* (Σ_1, Σ_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 (Σ, Σ') with a signature $(\Sigma, \Omega, \Sigma')$; and (2) the context may be strengthened by replacing Γ with a context Γ^+ 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 Ω leads to a more general notion of strengthening a signature:

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

We often use the less general notion (inserting a single Ω), 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* Σ, *a context* Γ' *is* stronger than Γ, *written* $\Sigma \vdash \Gamma' \leq_{\mathsf{ctx}} \Gamma$, *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 Σ in $\Sigma \vdash \mathcal{J}$ can be replaced by a stronger Σ', where $\mathcal{J}$ has the form A *type* or $s_1 \preceq s_2$ or $A \leq B$ or $c : A \to s$ or $A \sqsubset \tau$ or $c : C$. Lemma 9 says that $(\Sigma, \Omega, \Sigma')$ can replace (Σ, Σ') in $\Sigma, \Sigma'; S\langle K\rangle \vdash c : A \to s$ *safe at* t.Lemma 10 allows the sort t' in the judgment $\Sigma; S\langle K\rangle \vdash c : A \to s$ *safe at* t' to be replaced by a supersort t.

Using the above lemmas and Theorem 1, we can show that the key judgment "$\cdots c : A \to s$ safe" can be weakened by inserting Ω inside the signature:

Theorem 2 (Weakening 'safe').
If (Σ, Σ') *sig and* (Σ, Ω) *sig and* $\mathsf{dom}(\Sigma') \cap \mathsf{dom}(\Omega) = \emptyset$ *and* $\mathsf{dom}(\Sigma, \Omega, \Sigma') \cap S = \emptyset$ *and* K *does not mention anything in* $\mathsf{dom}(\Omega)$ *and* $S\langle K\rangle$ *preserves subsorting for* (Σ, Σ') *and* $(c : A \to s) \in K$ *and* $\Sigma, \Sigma'; S\langle K\rangle \vdash c : A \to s$ *safe then* $\Sigma, \Omega, \Sigma'; S\langle K\rangle \vdash c : A \to s$ *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 Σ is safely extended by Σ' and separately by Ω, then Ω and Σ', together, safely extend Σ.

Theorem 3 (Signature Interleaving).
If (Σ, Σ') *sig and* (Σ, Ω) *sig and* $\mathsf{dom}(\Sigma') \cap \mathsf{dom}(\Omega) = \emptyset$ *then* $(\Sigma, \Omega, \Sigma')$ *sig.*

Ultimately, we will show type preservation; in the preservation case for the 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 intersect function, we need to show that a stronger input to intersect yields a stronger output; that is, a longer (stronger) signature yields a stronger type B_+ (a subtype of B) and a stronger context Γ_+ typing as-variables.

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

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

Lemma 13 says that the result of 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 (Σ, Σ') *sig,* (Σ, Ω) *sig,* $\mathsf{dom}(\Sigma') \cap \mathsf{dom}(\Omega) = \emptyset$ *and* $\Sigma, \Omega, \Sigma' \vdash \Gamma^+ \leq_{\mathsf{ctx}} \Gamma$ *then*

(1) If $\Sigma, \Sigma'; \Gamma \vdash e : A$ *then* $\Sigma, \Omega, \Sigma'; \Gamma^+ \vdash e : A$*.*
(2) If $\Sigma, \Sigma'; \Gamma \vdash \theta : \Gamma'$ *then* $\Sigma, \Omega, \Sigma'; \Gamma^+ \vdash \theta : \Gamma'$*.*
(3) If $\Sigma, \Sigma'; \Gamma; p : A \vdash ms : D$ *then* $\Sigma, \Omega, \Sigma'; \Gamma^+; p : A \vdash ms : D$*.*

Properties of Values. Substitution properties (Lemmas 14 and 15) and inversion (or *canonical forms*) properties (Lemma 16) hold.

Type Preservation and Progress. The last important piece needed for type preservation is that intersect does what it says: if a value v matches p, then v has type B where B is one of the outputs of intersect.

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

The preservation result allows for a longer signature, to model entering the scope of a declare expression or the arms of a match. We implicitly assume that, in the given typing derivation, all types are well-formed under the local signature: for any subderivation of $\Sigma; \Gamma \vdash e' : B$, it is the case that $\Sigma \vdash B$ *type*.

Theorem 6 (Preservation).
If Σ *sig and* $\Sigma; \cdot \vdash e : A$ *and* $e \mapsto e'$ *then there exists* Σ' *such that* $\Sigma, \Sigma' \vdash e' : A$ *where* (Σ, Σ') *sig.*

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

7 Bidirectional Typing

The type assignment system in Fig. 6 is not syntax-directed, because the rules Sub and $\wedge$I apply to any shape of expression. Nor is the system directed by the syntax of types: rule 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 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. $\to$I, make the conclusion a checking judgment; in each elimination rule, e.g. 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 $\to$I becomes a checking judgment, because we know B from the conclusion.
5. Since the subsumption rule cannot synthesize, add rules such as Syn$\wedge$E$_1$, which were admissible in the type assignment system.

This yields the rules in Fig. 12. (Rules for the match typing judgment $\Sigma;\Gamma;p : A \vdash ms \Leftarrow B$ can be obtained from Fig. 10 by replacing ":" in "$e_1 : D$" and

$$\boxed{\Sigma;\Gamma \vdash e \Leftarrow A \quad \Sigma;\Gamma \vdash e \Rightarrow A}\ \text{Expr. } e \text{ checks against } (\Leftarrow) \text{ / synthesizes } (\Rightarrow) \text{ type } A$$

$$\frac{}{\Sigma;\Gamma,x : A,\Gamma' \vdash x \Rightarrow A}\,\mathsf{SynVar} \qquad \frac{\Sigma;\Gamma \vdash e \Rightarrow A \quad \Sigma \vdash A \leq B}{\Sigma;\Gamma \vdash e \Leftarrow B}\,\mathsf{ChkSub} \qquad \frac{A \in \vec{A} \quad \Sigma;\Gamma \vdash e \Leftarrow A}{\Sigma;\Gamma \vdash (e : \vec{A}) \Rightarrow A}\,\mathsf{SynAnno}$$

$$\frac{\Sigma;\Gamma,x : A \vdash e \Leftarrow B}{\Sigma;\Gamma \vdash (\lambda x.\, e) \Leftarrow (A \to B)}\,\mathsf{Chk}{\to}\mathsf{I} \qquad \frac{\Sigma;\Gamma \vdash e_1 \Rightarrow (A \to B) \quad \Sigma;\Gamma \vdash e_2 \Leftarrow A}{\Sigma;\Gamma \vdash e_1\, e_2 \Rightarrow B}\,\mathsf{Syn}{\to}\mathsf{E}$$

$$\frac{\Sigma;\Gamma \vdash v \Leftarrow A_1 \quad \Sigma;\Gamma \vdash v \Leftarrow A_2}{\Sigma;\Gamma \vdash v \Leftarrow (A_1 \wedge A_2)}\,\mathsf{Chk}{\wedge}\mathsf{I} \qquad \frac{\Sigma;\Gamma \vdash e \Rightarrow (A_1 \wedge A_2) \quad k \in \{1,2\}}{\Sigma;\Gamma \vdash e \Rightarrow A_k}\,\mathsf{Syn}{\wedge}\mathsf{E}_k$$

$$\frac{\Sigma;\Gamma \vdash e_1 \Leftarrow A_1 \quad \Sigma;\Gamma \vdash e_2 \Leftarrow A_2}{\Sigma;\Gamma \vdash (e_1, e_2) \Leftarrow A_1 * A_2}\,\mathsf{Chk}{*}\mathsf{I} \qquad \frac{}{\Sigma;\Gamma \vdash () \Leftarrow 1}\,\mathsf{Chk1I}$$

$$\frac{\Sigma \vdash c : A \to s \quad \Sigma;\Gamma \vdash e \Leftarrow A}{\Sigma;\Gamma \vdash c(e) \Leftarrow s}\,\mathsf{ChkDataI} \qquad \frac{\Sigma;\Gamma \vdash e \Rightarrow A \quad \Sigma;\Gamma;\_ : A \vdash ms \Leftarrow B}{\Sigma;\Gamma \vdash (\mathsf{case}\ e\ \mathsf{of}\ ms) \Leftarrow B}\,\mathsf{ChkDataE}$$

$$\frac{(\Sigma,\Sigma')\ \mathit{sig} \qquad \Sigma \vdash B\ \mathit{type} \qquad \Sigma,\Sigma';\Gamma \vdash e \Leftarrow B}{\Sigma;\Gamma \vdash (\mathsf{declare}\ \Sigma'\ \mathsf{in}\ e) \Leftarrow B}\,\mathsf{ChkDeclare}$$

Fig. 12. Bidirectional typing rules

"$\mathsf{ms} : \mathsf{D}$" with "$\Leftarrow$".) While this system is much more algorithmic than Fig. 6, the presence of intersection types requires backtracking: if we apply a function of type $(\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even})$, we need to synthesize $\mathsf{even} \to \mathsf{odd}$ first; if we subsequently fail (e.g. if the argument has type odd), we backtrack and try $\mathsf{odd} \to \mathsf{even}$. Similarly, if the signature contains several typings for a constructor c, we may need to try rule $\mathsf{ChkDataI}$ 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 let synthesizes a type for the bound expression). Rules with "more synthesis"—such as a synthesizing version of $*\mathsf{I}$—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 A. Rule $\mathsf{SynAnno}$ chooses one of these, backtracking if necessary. Multiple types may be needed if a λ-term is checked against intersection type: when checking $(\lambda x.\, x)$ against $(\mathsf{even} \to \mathsf{even}) \wedge (\mathsf{odd} \to \mathsf{odd})$, the type of x will be even inside the left subderivation of $\mathsf{Chk}\wedge\mathsf{I}$, but odd inside the right subderivation. Thus, if we annotate x with even, the check against $\mathsf{odd} \to \mathsf{odd}$ fails; if we annotate x with odd, the check against $\mathsf{even} \to \mathsf{even}$ 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 $\Gamma \vdash e \Leftarrow A$ *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 $e_{\Leftarrow}$ *such that* $|e_{\Leftarrow}| = e$ *and* $\Gamma \vdash e_{\Leftarrow} \Leftarrow A$.
(2) There exists $e_{\Rightarrow}$ *such that* $|e_{\Rightarrow}| = e$ *and* $\Gamma \vdash e_{\Rightarrow} \Rightarrow A$.

We also prove that the $\Rightarrow$ and $\Leftarrow$ judgments are decidable (Appendix, Theorem 10).

8 Related Work

Datasort Refinements. Freeman and Pfenning (1991) introduced datasort refinements with intersection types, defined the refinement restriction (where $A \wedge B$ 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 `%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* (Xi and Pfenning 1999; Xi 1998), 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 (Dunfield 2007a). More recent work in this vein, such as liquid types (Rondon et al. 2008), 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 Dunfield and Pfenning (2003) and Dunfield (2007b); 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 (Vazou et al. 2013) 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. Koot and Hage (2015) 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 λ-calculus interpreter could be refined with a sort `val` of values, while a lazy interpreter could be refined with a sort `whnf` of terms in weak head normal

form. If every val is a whnf, we might want to have $\mathsf{val} \preceq \mathsf{whnf}$. In the present system, these two refinements could be in separate `declare` blocks; in that case, val and whnf could not both be in scope, and the subsorting is not well-formed. Alternatively, one `declare` block could be nested inside the other. In that case, $\mathsf{val} \preceq \mathsf{whnf}$ 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 $\mathsf{val} \preceq \mathsf{whnf}$, 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 val and the block declaring 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 Davies (2005), 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 α-renames free variables—which would conventionally have the type $\mathsf{exp} \to \mathsf{exp}$—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 $\forall \alpha \preceq \mathsf{exp}.\ \alpha \to \alpha$, which works for any sort α below exp.

We would like to reason "backwards" from a pattern match over a polymorphic sort variable α. For example, if a value of type α matches the pattern $\mathsf{Plus}(\mathsf{x1}, \mathsf{x2})$, then we know that $\mathsf{Plus} : (\alpha_1 * \alpha_2) \to \alpha$ for some sorts α_1 and α_2. The recursive calls on $\mathsf{x1}$ and $\mathsf{x2}$ must preserve the property of being in α_1 and α_2, so $\mathsf{Plus}(\mathsf{f}\ \mathsf{x1}, \mathsf{f}\ \mathsf{x2})$ has type α, as needed. The mechanisms we have developed may be a good foundation for adding sort-bounded polymorphism: the $\mathsf{intersect}$ function would need to return a signature, as well as a context and type, so that the constructor typing $\mathsf{Plus} : (\alpha_1 * \alpha_2) \to \alpha$ can be made available.

Implementation. Currently, we have a prototype of a few pieces of the system, including a parser and implementations of the Σ *sig* judgment and the $\mathsf{intersect}$ 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.

References

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2008). https://gforge.inria.fr/frs/download.php/file/10994/tata.pdf. Accessed 18 Nov 2008

Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252 (1977)

Davies, R.: Practical refinement-type checking. Ph.D. thesis, Carnegie Mellon University, CMU-CS-05-110 (2005)
Davies, R.: SML checker for intersection and datasort refinements (pronounced "cider") (2013). https://github.com/rowandavies/sml-cidre
Davies, R., Pfenning, F.: Intersection types and computational effects. In: ICFP, pp. 198–208 (2000)
Dunfield, J.: Refined typechecking with stardust. In: Programming Languages Meets Program Verification (PLPV 2007) (2007a)
Dunfield, J.: A unified system of type refinements. Ph.D. thesis, Carnegie Mellon University, CMU-CS-07-129 (2007b)
Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: ICFP (2013). arXiv:1306.6032
Dunfield, J., Pfenning, F.: Type assignment for intersections and unions in call-by-value languages. In: Gordon, A.D. (ed.) FoSSaCS 2003. LNCS, vol. 2620, pp. 250–266. Springer, Heidelberg (2003). doi:10.1007/3-540-36576-1_16
Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: Principles of Programming Languages, pp. 281–292 (2004)
Freeman, T.: Refinement types for ML. Ph.D. thesis, Carnegie Mellon University, CMU-CS-94-110 (1994)
Freeman, T., Pfenning, F.: Refinement types for ML. In: Programming Language Design and Implementation, pp. 268–277 (1991)
Gentzen, G.: Untersuchungen über das logische Schließen. Mathematische Zeitschrift **39**, 176–210, 405–431 (1934). English translation, Investigations into logical deduction. In: Szabo, M. (ed.) Collected Papers of Gerhard Gentzen, pp. 68–131. North-Holland (1969)
Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. ACM **40**(1), 143–184 (1993)
Kennedy, A.: Programming languages and dimensions. Ph.D. thesis, University of Cambridge (1996)
Koot, R., Hage, J.: Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation, pp. 127–138 (2015)
Lovas, W.: Refinement types for logical frameworks. Ph.D. thesis, Carnegie Mellon University, CMU-CS-10-138 (2010)
Pfenning, F., Schürmann, C.: System description: Twelf—a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999). doi:10.1007/3-540-48660-7_14
Pientka, B., Dunfield, J.: Beluga: a framework for programming and reasoning with deductive systems (system description). In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 15–21. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14203-1_2
Poswolsky, A., Schürmann, C.: System description: Delphin–a functional programming language for deductive systems. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008). Electronic Notes in Theoretical Computer Science, vol. 228, pp. 135–141 (2009)
Reynolds, J.C.: Types, abstraction, and parametric polymorphism. In: Information Processing 83, pp. 513–523. Elsevier (1983). http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/typesabpara.pdf
Reynolds, J.C.: Design of the programming language Forsythe. Technical report CMU-CS-96-146, Carnegie Mellon University (1996)

Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: Programming Language Design and Implementation, pp. 159–169 (2008)

Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 209–228. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37036-6_13

Wright, A.K.: Simple imperative polymorphism. Lisp Symbolic Comput. **8**(4), 343–355 (1995)

Xi, H.: Dependent types in practical programming. Ph.D. thesis, Carnegie Mellon University (1998)

Xi, H., Pfenning, F.: Dependent types in practical programming. In: Principles of Programming Languages, pp. 214–227 (1999)

Zeilberger, N.: The logical basis of evaluation order and pattern-matching. Ph.D. thesis, Carnegie Mellon University, CMU-CS-09-122 (2009)