Sum types in Futhark

Robert Schenck

December 9, 2019

Advisors:
Martin Elsman
Troels Henriksen

Abstract

While a common staple of many modern functional programming languages, sum types are an unusual feature in specialized, compute-oriented languages. We advocate that sum types have merit even in this restricted setting: providing increased safety and better abstraction at manageable cost.

This thesis explores adding sum types to Futhark—a compute-oriented, purely functional, data-parallel array programming language. The work includes a theoretical development of sum types in model type system closely related to Futhark's type system, implementation details for adding sum types to the Futhark compiler, and an analysis of the implementation—including suggestions to improve performance.

Contents

1 Introduction **2**

2 Theory **4**
2.1 Syntax . . . 4
2.2 Evaluation . . . 6
2.3 A monomorphic type system . . . 7
2.4 Don't go wrong! . . . 9
2.5 Constraint-based typing . . . 14

3 Implementation **22**
3.1 Overview . . . 22
3.2 Syntax . . . 22
3.3 Type checking . . . 24
3.4 Pattern exhaustivity . . . 25
3.5 Internalization . . . 27

4 Evaluation and improvements **31**
4.1 Problematic arrays of sum types . . . 31
4.2 Constructor deduplication . . . 31
4.3 Efficiency of generated conditional statements . . . 32
4.4 Sum type polymorphism . . . 35
4.5 The utility of sum types in Futhark . . . 36

5 Conclusions and future work **38**

Bibliography **39**

Chapter 1

Introduction

This thesis covers the design and implementation of sum types in the Futhark[1] programming language. Futhark is a functional high-performance data-parallel array language: its intended purpose is to be used to accelerate compute-intensive parallel tasks [HSE+17]. Sum types are an unusual feature of similar languages. Notably, the type system of the NESL [Ble95] data-parallel programming language is somewhat spartan (with no support for sum types) and Accelerate—an embedded data-parallel language in the Haskell programming language—limits the types of array elements to primitive types [CKL+11]. On the other hand, the Data Parallel Haskell project implemented nested data-parallel programming into the Glasgow Haskell Compiler (GHC) with support for Haskell's rich type system, including sum types [CLPJ+07]. Unfortunately, work on the project ceased in 2010 and the existing implementation work was ultimately removed from GHC.[2]

Sum types are a staple of modern functional programming languages with good reason: programs dealing with heterogeneous collections of data arise often and type- and expression-level features to deal with such collections are a powerful abstraction for the programmer. We believe that sum types are also merited in programming languages like Futhark. In our experience, rewriting Futhark programs to use sum types results in simpler, more expressive programs that make good use of Futhark's type checker (see Section 4.5 for a discussion).

Opponents of sum types in a computational language like Futhark may have performance concerns. While the implementation described in this thesis isn't particularly efficient, a number of possible improvements are discussed in Chapter 4. With sufficient (and realistic) optimization the cost of sum types can be made sufficiently small for pragmatic use. Of course, sum types may be used judiciously by the programmer and can be omitted where performance and efficiency is critical.

Contributions

This thesis contributes the following:

1. A formal study of sum types in a model language based off of Futhark, including a proof of soundness (Chapter 2).

[1] https://futhark-lang.org/

[2] See the discussion at https://gitlab.haskell.org/ghc/ghc/wikis/data-parallel

2. The addition of sum type support to the Futhark compiler (Chapter 3), consisting of three main parts:
   (a) An extension of Futhark's syntax with sum types, constructor expressions, and match expressions as well as expanding Futhark's pattern syntax (Section 3.2).
   (b) Augmentation of Futhark's type system to support sum types (Section 3.3, Section 3.4).
   (c) Transformation logic to convert sum types, constructor expressions, and match expressions into Futhark's intermediate representation language (Section 3.5).
3. An analysis of the implementation, including suggestions to improve efficiency as well as case studies of example Futhark programs which benefit from sum types.

Chapter 2

Theory

This chapter is a formal study of sum types in a language modeled after Futhark. We begin by defining some basics of the language before introducing a monomorphic type system for the language and proving a number of properties about it. We subsequently study a constraint-based version of the type system that closely resembles Futhark's type system. As the language and type system are good facsimiles of Futhark's respective counterparts (as far as sum types are concerned, at least) this theoretical analysis is important in that it informs us about the correctness of the actual implementation.

2.1 Syntax

2.1.1 Types

For the theoretical development, we describe a simple λ-calculus, with syntax closely related to a subset of Futhark's syntax. Let *id* range over a set of labels $\mathcal{L}$ and let t range over a set of built-in types, which includes at least the unit type `unit`.[3] Types are defined by the grammar

$$
\begin{array}{rcll}
\tau & ::= & t & \text{built-in type} \\
 & | & \tau_1 \rightarrow \tau_2 & \text{function} \\
 & | & (\tau_1, \tau_2) & \text{tuple} \\
 & | & \#id_1\ \tau_1 \mid \cdots \mid \#id_n\ \tau_n & \text{sum type}
\end{array}
$$

We let $\mathcal{T}$ be the set of all types, ranged over by τ, σ, and ψ in the discussion that follows.

Sum types

Of special interest are sum types, which are composed of *clauses* separated by a pipe; each clause consists of a #-prefixed identifier—the *constructor*—followed by a type *field*, which delineates the type of the payload of the constructor. Each constructor has precisely one payload field. A sum type with two distinct clauses with the same constructor is illegal. The ordering of the clauses does not matter: two sum types

[3] Which adds supports for enumerations, enabling the calculus to represent conditionals via matches on the type `#true ()` | `#false ()`.

consisting of different permutations of the same clauses are the same type. Two sum types are disjoint if they share no constructors in common. The clauses of two disjoint sum types may be merged to form a larger sum type using the $|$ operator.

2.1.2 Terms

Let x, y, z range over the set of variables $\mathcal{V}$ and let c range over the set of expression constants $\mathcal{C}$, which includes at least the unit value $(())$. Let $\mathcal{E}$ be the set of terms, which are defined by the grammar[4]

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & c & \text{constant} \\
 & | & e : \tau & \text{type ascription} \\
 & | & (e_1, e_2) & \text{tuple} \\
 & | & \lambda x : \tau .\, e & \text{function} \\
 & | & e_1 e_2 & \text{application} \\
 & | & \#id\ e & \text{constructor} \\
 & | & \textbf{match}\ e\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n & \text{match}
\end{array}
$$

where p is a *pattern*, described in Section 2.1.4. The presence of type ascriptions in a minimal language like this may seem odd: they are in fact a necessary component of well-typed constructor terms, as we'll see later in the development. We also assume the existence of a function $const : \mathcal{E} \to \mathcal{T}$ which returns the type of a constant term; e.g., $const(()) = \texttt{unit}$.

2.1.3 Substitutions

A *term substitution* is a partial mapping from term variables to terms. For example, we write a term substitution as $[x \mapsto e_1, y \mapsto e_2]$, which maps x to e_1 and y to e_2. Substitutions always substitute all variables at once, i.e., $[x \mapsto ye_1, y \mapsto e_2](xy) = ye_1e_2$, not $e_2e_1e_2$. The substitution operation always does any necessary renaming to avoid variable capture/release (we leave these specifics unspecified). A substitution S works as one would expect on terms:

$$
\begin{aligned}
S(x) &= \begin{cases} v & \text{if } (x \mapsto v) \in S \text{ for some } v \\ x & \text{otherwise} \end{cases} \\
S(c) &= c \\
S(e : \tau) &= S(e) : \tau \\
S(e_1, e_2) &= (S(e_1), S(e_2)) \\
S(\lambda x : \tau .\, e) &= \lambda x : \tau .\, S(e) \\
S(e_1 e_2) &= S(e_1) S(e_2) \\
S(\#id\ e) &= \#id\ S(e) \\
S(\textbf{match}\ e\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n) &= \textbf{match}\ S(e)\ \textbf{case}\ p_1 \to S(e_1) \cdots \textbf{case}\ p_n \to S(e_n)
\end{aligned}
$$

[4]Note the lack of a **let**-expression: $\textbf{let}\ p = e_1\ \textbf{in}\ e_2$ is equivalent to $\textbf{match}\ e_1\ \textbf{case}\ p \to e_2$. This is actually safer in general since a failed pattern match can be guarded against by augmenting matches with a catchall case ($\textbf{case}\ x \to e$).

In the case of multiple mappings for the same variable, the right-most mapping of a substitution takes precedence: $[x \mapsto e_1, x \mapsto e_2]x = e_2$. Note that the empty substitution $[]$ is the identity substitution.

2.1.4 Patterns

Patterns are defined by the grammar[5]

$$p ::= c \mid x \mid (p_1, p_2) \mid \#id\ p$$

Note that a given variable x may not be repeated in a pattern (all variables in a pattern must be distinct). Along with patterns, we define a *pattern match* operator

$$? : \mathcal{P} \to \mathcal{E} \to \mathcal{S} \cup \{\bot\}$$

where $\mathcal{P}$ is the set of patterns and $\mathcal{S}$ is the set of term substitutions. We define the operator piece-wise:[6]

$$\begin{aligned}
c_p \ ?\ c_e &= \begin{cases} [] & c_p = c_e \\ \bot & \text{otherwise} \end{cases} \\
x \ ?\ e &= [x \mapsto e] \\
(p_1, p_2) \ ?\ (e_1, e_2) &= \begin{cases} S_1 \cup S_2 & S_1 = p_1 \ ?\ e_1,\ S_2 = p_2 \ ?\ e_2 \\ & S_1 \neq \bot, \text{ and } S_2 \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
\#id_p\ p \ ?\ \#id_e\ e &= \begin{cases} p \ ?\ e & \#id_p = \#id_e \\ \bot & \text{otherwise} \end{cases} \\
p \ ?\ e &= \bot
\end{aligned}$$

Intuitively, ? takes a pattern p and an expression e to match that pattern on and either says that the match failed ($p\ ?\ e = \bot$) or returns a substitution S with the bindings that the match generated.

2.2 Evaluation

Values

Evaluation is call-by-value and left-to-right. To formalize this, we first define *values*:

$$v \quad ::= c \mid \lambda x \,.\, e \mid \#id\ v \mid (v_1, v_2) \mid v : \tau$$

Values are simply terms which are *fully evaluated* (i.e., cannot be simplified further via a step of computation) and *closed* (have no free variables).

[5] Patterns often also include the wildcard pattern (_), which acts as a non-binding variable. We omit wildcards in the theoretical development as they're unnecessary (a variable can be used and the binding can be discarded) and would complicate constructing typing judgments on patterns (see Section 2.3.2).

[6] When expanding the ? operator, the definition should be read top-to-bottom until a match on the LHS is found with the current usage.

2.2.1 Operational semantics

Primitive reduction

We define a *primitive reduction* relation $\rightsquigarrow_p$ which describes a step of computation in the language. In our case, that consists of either a) function application or b) match reduction. These reduction rules are shown in Figure 2.1 below.

$$\frac{}{(\lambda x : \tau . e)v \rightsquigarrow_p [x \mapsto v]e} \text{ S-APP}$$

$$\frac{p_i \mathbin{?} v = S_i \qquad i \text{ is the smallest integer with } S_i \neq \bot}{\textbf{match } v \textbf{ case } p_1 \rightarrow e_1 \cdots \textbf{case } p_n \rightarrow e_n \rightsquigarrow_p S_i(e_i)} \text{ S-MATCH}$$

Figure 2.1: Primitive reductions for function application and **match**-expressions.

Evaluation contexts

In order to constrain where reductions may take place and in what order we define *evaluation contexts*:

$$E \quad ::= \quad \bullet \mid vE \mid Ee \mid (v, E) \mid (E, e) \mid \#id\ E : \tau \mid E : \tau \mid \textbf{match } E \textbf{ case } p_1 \rightarrow e_1 \cdots \textbf{case } p_n \rightarrow e_n$$

An evaluation context E may be thought of as an expression-building function $E : \mathcal{E} \rightarrow \mathcal{E}$. It accepts an expression and places it within a larger expression. Each evaluation context has exactly one *hole* ($\bullet$) which is substituted for the expression that the context is applied on. We write $E[e]$ to replace E's hole with e. More precisely, $E[e] = [\bullet \mapsto e]E$.

Augmenting primitive reduction ($\rightsquigarrow_p$), we define a general reduction relation $\rightsquigarrow$ which describes a step of computation for any reducible term in the language. The only "real" computation is defined by the primitive reductions of Figure 2.1. All other reductions are simply contextual primitive reductions, a notion captured by the *context rule*:

$$\frac{e \rightsquigarrow_p e'}{E[e] \rightsquigarrow E[e']} \text{ CTX}$$

With CTX, evaluation contexts enforce the call-by-value, left-right semantics of the language.

2.3 A monomorphic type system

In this section, we'll develop a simple monomorphic type system for our language and prove some standard properties about it. In Section 2.5, we'll augment the system with Hindley-Damas constraint-based typing that is closely related to Futhark's type checker.

2.3.1 Typing relation

A *typing context* Γ is a sequence of term variables and their types. A *type relation* is a three-place relation

$$\Gamma \vdash e : \tau$$

which asserts that term e has type τ under the typing context Γ. As usual, we use a comma to extend the environment: $\Gamma, x : \tau$ is the context Γ extended with a variable x (of type τ). A typing relation with the empty context is written with $\vdash e : \tau$.

2.3.2 Patterns

Binding pattern variables

All variables appearing in patterns are free.[7] This poses a difficulty when typing match statements: how can we (correctly) assign the variables in a pattern types? We must do so in order to a) answer whether a pattern makes sense given the term being matched upon and b) determine the type of the body of a case expression.

To address this issue, we define a function

$$bind : \mathcal{P} \times \mathcal{T} \to \{x : \tau \mid x \in \mathcal{V}, \tau \in \mathcal{T}\}$$

that generates variable-typing pairs for pattern variables:

$$bind(p, \tau) = \begin{cases} \{x : \tau\} & p = x \\ bind(p_1, \tau_1) \cup bind(p_2, \tau_2) & p = (p_1, p_2),\ \tau = (\tau_1, \tau_2) \\ bind(p', \tau_i) & p = \#id_p\ p',\ \tau = \#id_1\ \tau_1 \mid \cdots \mid \#id_n\ \tau_n,\ id_p = id_i \\ \varnothing & \text{otherwise} \end{cases}$$

Since patterns are defined by a grammar that is just a subset of the terms grammar—using *bind*—we can massage the typing relation to make sense on patterns:

$$bind(p, \tau) \vdash p : \tau$$

This also explains why *bind* doesn't include a "failure" result like the pattern match operator ? does: if a pattern fails to match the type passed to *bind*, it will be impossible to construct a type assertion featuring the pattern and the given type (and *bind* will just return the empty set).

Exhaustivity

We need to ensure that matches don't get stuck due to inexhaustive patterns. We define a predicate $exhaustive(P, \tau)$ which takes a set of patterns P and a type τ and is true if and only if for all contexts Γ and terms e with $\Gamma \vdash e : \tau$, there exists $p \in P$ with $p\ ?\ e \neq \bot$. The exact definition of *exhaustive* depends on the built-in types of the language; a discussion of how this check is done for Futhark can be found in Section 3.4.

2.3.3 Typing rules

Figure 2.2 gives the typing rules for the language. Note that the conclusion of the CONSTR rule enforces a type ascription[8]; without this ascription, there would be no way to determine the type of constructors without a more complex typing system (e.g.,

[7] This follows from the semantics defined by the S-MATCH rule and the definition of the ? operator.

[8] This also has the side effect that it's possible to choose the wrong rule when typing a constructor, namely ASCRIP, and simply get stuck as a result. The ASCRIP rule is entirely unnecessary (except for type documentation purposes)—it's kept to provide compatibility with the constraint-based rules developed in Section 2.5.

as in Section 2.5). A single constructor only gives us information about a single clause of a sum type: there may be arbitrarily more of them, of which we know nothing.

$$\frac{}{\Gamma \vdash c : \mathit{const}(c)}\ \textsc{Const} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e : \tau) : \tau}\ \textsc{Ascrip}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}\ \textsc{Tuple} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) : \tau_1 \to \tau_2}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_2 \text{ is a sum type with the clause } \#\mathit{id}\ \tau_1}{\Gamma \vdash (\#\mathit{id}\ e_1 : \tau_2) : \tau_2}\ \textsc{Constr}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \qquad \mathit{exhaustive}(\cup_{i=1}^{n}\{p_i\}, \tau_1) \\ \text{for } i \in 1 \ldots n\text{:} \qquad \mathit{bind}(p_i, \tau_1) \vdash p_i : \tau_1 \qquad \Gamma \cup \mathit{bind}(p_i, \tau_1) \vdash e_i : \tau_2\end{array}}{\Gamma \vdash (\textbf{match}\ e\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n) : \tau_2}\ \textsc{Match}$$

Figure 2.2: Monomorphic typing rules.

The $\mathit{bind}(p_i, \tau) \vdash p_i : \tau$ premise of Match ensures that each pattern p_i can be typed against the type of the expression being matched on, i.e., the pattern structurally matches the expression at the type level.

2.4 Don't go wrong!

In this section, we'll prove some basic properties about the type system presented in the previous section. We want to make sure that the language "doesn't go wrong" in the sense that as long as an expression is a well-typed, we have a guarantee that we can evaluate said expression without things going awry. This is generally referred to as *safety* or *soundness* and boils down to two sub-properties: *progress* and *preservation*. We begin with the progress property.

2.4.1 Progress

Theorem 2.1 formally captures the notion that well-typed terms do not get "stuck". That is, for any closed well-typed term of our language, either that term is fully evaluated (it's a value) or we can take a step of computation and reduce the term.

Theorem 2.1 (Progress)**.** If $\vdash e : \tau$ then e is a value or there exists an e' such that $e \leadsto e'$.

Proof. By induction on $\vdash e : \tau$.

Case Const: $e = c$, which is a value.

Case Var: $\qquad e = x \qquad x : \tau \in \varnothing$

$x : \tau \in \varnothing$ implies falsity.

Case ASCRIP: $e = e' : \tau \qquad \vdash e' : \tau$

Invoking the inductive hypothesis on $\vdash e' : \tau$, we have two cases to consider:

[$e' = v'$]: $v' : \tau$ is a value.

[e' is not a value and $\exists e''. e' \rightsquigarrow e''$]: By inversion on CTX, there exists E, e'_p, e''_p such that $e'_p \rightsquigarrow_p e''_p$ and $e' = E[e'_p], e'' = E[e''_p]$. By the definition of evaluation contexts and CTX, $(E : \tau)[e'_p] \rightsquigarrow (E : \tau)[e''_p]$, which simplifies to $(e' : \tau) \rightsquigarrow (e'' : \tau)$.

Case TUPLE:

$$e = (e_1, e_2) \qquad \tau = (\tau_1, \tau_2) \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2$$

(For the remainder of the proof, the inductive hypothesis will be invoked implicitly as required.)

[$e_1 = v_1, e_2 = v_2$]: $e = (v_1, v_2)$ is a value.

[$e_1 = v_1$, e_2 is not a value and $\exists e'_2. e_2 \rightsquigarrow e'_2$]:
Inversion of CTX gives us $E, e_{2,p}, e'_{2,p}$ such that $e_2 = E[e_{2,p}]$ and $e'_2 = E[e'_{2,p}]$ and $e_{2,p} \rightsquigarrow_p e'_{2,p}$. By CTX, $(v_1, E)[e_{2,p}] \rightsquigarrow (v_1, E)[e'_{2,p}]$, which can be rewritten as $(v_1, e_2) \rightsquigarrow (v_1, e'_2)$.

[e_1, e_2 are not values and $\exists e'_1. e_1 \rightsquigarrow e'_1$]: Analogous to the previous case, using (E, e_2) as the evaluation context.

Case FUN: $e = \lambda x : \tau_1. e'$, which is a value.

Case APP:

$$e = e_1 e_2 \qquad \tau = \tau_2 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \rightarrow \tau_2$$

[$e = v_1 v_2$]: Since $\Gamma \vdash e_2 : \tau_1 \rightarrow \tau_2$, by FUN, $v_1 = \lambda x : \tau_1. e_1$ for some e_1. Hence, by S-APP and CTX (choosing $E = \bullet$), $(\lambda x : \tau_1. e_1) v_2 \rightsquigarrow [x \mapsto v_2] e_1$.

[$e = v_1 e_2$ and $\exists e'_2. e_2 \rightsquigarrow e'_2$]: Analogous to the second case of the TUPLE case.

[e_1, e_2 are not values and $\exists e'_1. e_1 \rightsquigarrow e'_1$]: Analogous to the third case of the TUPLE case.

Case CONSTR:

$$e = \#id\ e_1 : \tau_2 \qquad \tau = \tau_2 \qquad \tau_2 \text{ is a sum type with the clause } \#id\ \tau_1 \qquad \Gamma \vdash e_1 : \tau_1$$

[$e_1 = v_1$]: $\#id\ v_1 : \tau_2$ is a value.

[e_1 is not a value and $\exists e'_1. e_1 \rightsquigarrow e'_1$]: Analogous to ASCRIP with the context $\#id\ E$.

Case MATCH:

$$e = \textbf{match}\ e'\ \textbf{case}\ p_1 \rightarrow e_1 \cdots \textbf{case}\ p_n \rightarrow e_n \qquad \Gamma \vdash e' : \tau_1$$

$$exhaustive(\cup_{i=1}^{n} \{p_i\}, \tau_1)$$

[$e' = v'$]: The exhaustivity of the patterns guarantees that there is an i with $p_i\ ?\ v' = S_i \neq \perp$. Hence, by S-MATCH and CTX,

$$\textbf{match}\ v'\ \textbf{case}\ p_1 \rightarrow e_1 \cdots \textbf{case}\ p_n \rightarrow e_n \rightsquigarrow S_i(e_i)$$

[e' is not a value and $\exists e''. e' \rightsquigarrow e''$]: Analogous to the ASCRIP case with context **match** E **case** $p_1 \to e_1 \cdots$ **case** $p_n \to e_n$.

□

2.4.2 Preservation

The preservation property states that the $\rightsquigarrow$ relation preserves the well-typedness of terms. That is, if a closed term e has type τ before a step of computation, it should also have the same type τ after a step of computation.

Contextual typing

Proving preservation requires a bit more work due to the use of evaluation contexts. Specifically, while it is straightforward to prove preservation for primitive steps ($\rightsquigarrow_p$), we need a mechanism that allows us to assert the typing properties of a context so that we can prove preservation for contextual steps ($\rightsquigarrow$). We follow [DJK+18] here and introduce a *contextual typing* relation $\vdash E : \psi \Rightarrow \tau$ in Figure 2.3, which asserts that when E's hole is filled with an expression e with $\vdash e : \psi$, then $\vdash E[e] : \tau$.

$$\frac{}{\vdash \bullet : \tau \Rightarrow \tau}\ \textsc{Hole} \qquad \frac{\vdash e : \tau_1 \qquad \vdash E : \psi \Rightarrow \tau_1 \to \tau_2}{\vdash Ee : \psi \Rightarrow \tau_2}\ \textsc{L-App}$$

$$\frac{\vdash v : \tau_1 \to \tau_2 \qquad \vdash E : \psi \Rightarrow \tau_1}{\vdash vE : \psi \Rightarrow \tau_2}\ \textsc{R-App} \qquad \frac{\vdash e : \tau_2 \qquad \vdash E : \psi \Rightarrow \tau_1}{\vdash (E, e) : \psi \Rightarrow (\tau_1, \tau_2)}\ \textsc{L-Tuple}$$

$$\frac{\vdash v : \tau_1 \qquad \vdash E : \psi \Rightarrow \tau_2}{\vdash (v, E) : \psi \Rightarrow (\tau_1, \tau_2)}\ \textsc{R-Tuple}$$

$$\frac{\vdash E : \psi \Rightarrow \tau_1 \qquad \tau_2 \text{ contains the clause } \#id\ \tau_1}{\vdash (\#id\ E : \tau_2) : \psi \Rightarrow \tau_2}\ \textsc{E-Constr}$$

$$\frac{\vdash E : \psi \Rightarrow \tau}{\vdash (E : \tau) : \psi \Rightarrow \tau}\ \textsc{E-Ascrip}$$

$$\frac{\begin{array}{c}\vdash E : \psi \Rightarrow \tau_1 \qquad \mathit{exhaustive}(\cup_{i=1}^{n}\{p_i\}, \tau_1) \\ \text{for } i \in 1 \ldots n: \qquad \mathit{bind}(p_i, \tau_1) \vdash p_i : \tau_1 \qquad \Gamma \cup \mathit{bind}(p_i, \tau_1) \vdash e_i : \tau_2\end{array}}{\vdash \textbf{match}\ E\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n : \psi \Rightarrow \tau_2}\ \textsc{E-Match}$$

Figure 2.3: Contextual typing rules.

With the contextual typing relation in hand, we can now state and prove a lemma that allows us pick out a well-typed expression from its evaluation context.

Lemma 2.2. If $\vdash E[e] : \tau$ then there exists a type ψ such that $\vdash E : \psi \Rightarrow \tau$ and $\vdash e : \psi$.

Proof. By induction on E.

Case [$E = \bullet$]: Simply choose $\psi = \tau$.

Case $[E = vE']$: By inversion of APP, we have $\vdash v_1 : \tau_1 \to \tau_2$ and $\vdash E'[e] : \tau_1$. By the inductive hypothesis, there's some type ψ such that $\vdash E' : \psi \Rightarrow \tau_1$ and $\vdash e : \psi$. By R-APP, $\vdash vE' : \psi \Rightarrow \tau_2$.

Case $[E = E'e']$: By inversion of APP, we have $\vdash E'[e] : \tau_1 \to \tau_2$ and $\vdash e' : \tau_1$. By the inductive hypothesis, there's some type ψ such that $\vdash E' : \psi \Rightarrow \tau_1 \to \tau_2$ and $\vdash e : \psi$. By L-APP, $\vdash E'e' : \psi \Rightarrow \tau_2$.

Case $[E = (v, E')]$: By inversion of TUPLE, $\vdash v : \tau_1$ and $\vdash E'[e] : \tau_2$. By the inductive hypothesis, we have ψ such that $\vdash E' : \psi \Rightarrow \tau_2$ and $\vdash e : \psi$. By R-TUPLE, $\vdash (v, E') : \psi \Rightarrow (\tau_1, \tau_2)$.

Case $[E = (E', e)]$: Similar to the $E = (v, E')$ case.

Case $[E = \#id\ E' : \tau]$: By inversion of CONSTR, $\vdash E'[e] : \tau_1$; hence, τ must have a clause $\#id\ \tau_1$. By the inductive hypothesis, $\vdash E' : \psi \Rightarrow \tau_1$ and $\vdash e : \psi$ for some type ψ. By E-CONSTR, $\vdash (\#id\ E' : \tau) : \psi \Rightarrow \tau$, as desired.

Case $[E = \textbf{match}\ E\ '\textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n]$: Inversion of MATCH yields

$$\Gamma \vdash E'[e] : \tau_1 \qquad\qquad exhaustive(\cup_{i=1}^{n}\{p_i\}, \tau_1)$$

$$\text{for } i \in 1 \ldots n: \qquad bind(p_i, \tau_1) \vdash p_i : \tau_1 \qquad \Gamma \cup bind(p_i, \tau_1) \vdash e_i : \tau_2$$

By the inductive hypothesis, we have an ψ such that $\vdash E' : \psi \Rightarrow \tau$ and $\vdash e : \psi$. By E-MATCH, $\vdash \textbf{match}\ E\ '\textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n : \psi \Rightarrow \tau_2$.

□

We continue with a lemma that shows that the contextual typing relation behaves as we intend.

Lemma 2.3. If $\vdash E : \psi \Rightarrow \tau$ and $\vdash e : \psi$ then $\vdash E[e] : \tau$.

Proof. By induction on $\vdash E : \psi \Rightarrow \tau$. All cases (except for HOLE, which follows immediately by the assumptions) proceed by invoking the inductive hypothesis on the context E' which appears in the premise of the given contextual typing rule to obtain a typing $\vdash E'[e] : \tau'$. This typing can be used to directly construct the required typing assertion via the typing rules in Figure 2.2.

□

We also need a number of results about the typing of expressions under substitution as well as the typing of expressions evaluated under bindings produced by pattern matches.

Lemma 2.4. If $\Gamma, x : \tau_1 \vdash e : \tau_2$ and $\Gamma \vdash v : \tau_1$ then $\Gamma \vdash [x \mapsto v]e : \tau_2$.

Proof. By induction on $\Gamma, x : \tau_1 \vdash e : \tau_2$. For each inductive rule, we use the inductive hypothesis on the rule's premises to obtain valid typings with the substitution applied. The rule itself can then be used to construct $\Gamma \vdash [x \mapsto v]e : \tau_2$, as required. □

Corollary 2.5. If $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \psi$ and $\Gamma \vdash v_1 : \tau_1, \ldots, \Gamma \vdash v_n : \tau_n$, then $\Gamma \vdash [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]e : \psi$.

Proof. By induction on n, using Lemma 2.4. □

Lemma 2.6. If $p \mathbin{?} e = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] = S$ and $\Gamma \vdash e : \tau$, then, for each $i \in \{1 \ldots n\}$, there exists τ_i such that $\Gamma \vdash e_i : \tau_i$.

Proof. By induction on p.

Case $[p = c]$: $S = \varnothing$, so the statement is vacuously true.

Case $[p = x]$: $S = [x \mapsto e]$. We need to find a τ' such that $\Gamma \vdash e : \tau'$, so we just choose $\tau' = \tau$.

Case $[p = (p', p'')]$: By the definition of ?, $e = (e', e'')$ where $p' \mathbin{?} e' = S'$ and and $p'' \mathbin{?} e'' = S''$. By inversion of TUPLE, $\tau = (\tau', \tau'')$, $\Gamma \vdash e' : \tau'$ and $\Gamma \vdash e'' : \tau''$. Let $S' = [x'_1 \mapsto e'_1, \ldots, x'_r \mapsto e_r]$ and $S'' = [x''_1 \mapsto e''_1, \ldots, x''_s \mapsto e''_s]$. By the inductive hypothesis, there exists τ'_j and τ'_k such that $\Gamma \vdash e'_j : \tau'_j$ and $\Gamma \vdash e''_k : \tau''_k$ for each $j \in \{1, \ldots, r\}$ and $k \in \{1, \ldots, s\}$. Since $S = S' \cup S''$, we're done.

Case $[p = \#id\ p']$: Immediate by the inductive hypothesis.

□

Corollary 2.7. If $p \mathbin{?} e = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] = S$ and $\Gamma \vdash e : \tau_1$ and $\Gamma \cup bind(p, \tau) \vdash e : \tau$, then $\Gamma \vdash S(e) : \tau_2$.

Proof. By Lemma 2.6, for each $i \in \{1 \ldots n\}$, there exists τ_i such that $\Gamma \vdash e_i : \tau_i$. By Corollary 2.5 and the definition of *bind*, $\Gamma \vdash S(e) : \tau_2$. □

Finally, we move on to the meat of the matter: preservation. Since all computation boils down to function application and match reduction, the crux of the result hinges on proving preservation under primitive reduction. Preservation under general reduction follows readily thereafter.

Lemma 2.8 (Primitive preservation)**.** If $\vdash e : \tau$ and $e \rightsquigarrow_p e'$ then $\vdash e' : \tau$.

Proof. By induction on $e \rightsquigarrow_p e'$.

Case S-APP:

$$e = (\lambda x : \tau_1 . e'')v \qquad \tau = \tau_2 \qquad e' = [x \mapsto v]e''$$

By inversion on $\vdash (\lambda x : \tau_1 . e'')v : \tau$ (via APP), we have $\vdash \lambda x : \tau_1 . e'' : \tau_1 \to \tau_2$ and $\vdash v : \tau_1$. By inversion (via FUN), $x : \tau_1 \vdash e'' : \tau_2$. By Lemma 2.4, $\vdash [x \mapsto v]e'' : \tau_2$.

Case S-MATCH:

$$p_i \mathbin{?} v = S_i \qquad e = \textbf{match}\ v\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n \qquad \tau = \tau_2 \qquad e' = S_i(e_i)$$

By inversion on $\vdash e = \textbf{match}\ v\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n : \tau_2$ (via MATCH), we have $\vdash v : \tau_1$ and $bind(p_i) \vdash e_i : \tau_2$ for each $i \in 1 \ldots n$. By Corollary 2.7, $\vdash S_i(e_i) : \tau_2$.

□

Theorem 2.9 (Preservation)**.** If $\vdash e : \tau$ and $e \rightsquigarrow e'$ then $\vdash e' : \tau$.

Proof. By inversion on $e \rightsquigarrow e'$ (via CTX), there exists E, e_p, e'_p such that $e_p \rightsquigarrow e'_p$, $e = E[e_p]$, and $e' = E[e'_p]$. By Lemma 2.2, there exists a type ψ such that $\vdash E : \psi \Rightarrow \tau$ and $\vdash e_p : \psi$. By Lemma 2.8, $\vdash e'_p : \psi$. By Lemma 2.3, $\vdash e' : \tau$. □

Corollary 2.10. If $\vdash e : \tau$ and $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_n$ then $\vdash e_n : \tau$.

Proof. Straightforward induction on n (with $n = 2$ as the base case). □

2.4.3 Soundness

We now have all the necessary results to prove our goal: things shouldn't go wrong!

Theorem 2.11 (Soundness)**.** If $\vdash e_1 : \tau$ and $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_n$, then e_n is a value or there exists e_{n+1} such that $e_n \rightsquigarrow e_{n+1}$.

Proof. By induction on n. The base case (with $n = 2$) is immediate by Theorem 2.1. For the inductive case, the inductive hypothesis gives us that e_n is either a value or can take another step of computation. By Corollary 2.10, $\vdash e_n : \tau$ and, by Theorem 2.1, we're done. □

2.5 Constraint-based typing

In this section, we augment the monomorphic type system of Section 2.3 into a constraint-based system that more closely matches Futhark's type system with respect to its type checking/inference algorithms.

The type system is based on the Damas-Milner [DM82] (also known as Hindley-Milner) type system. The system uses *type variables* and *constraints* on these variables to resolve a term's type. Note that this is distinct from the type variables in a calculus like System F in that the type variables are strictly introduced by the type checking mechanisms and are never bound.[9] To put things simply: type variables never appear in programs.[10]

2.5.1 Type variables

The first thing to do is extend the types from Section 2.1.1 with type variables:

$$\begin{array}{rcll} \tau & ::= & \cdots & \\ & | & \alpha & \text{type variable} \end{array}$$

We let α, β and γ range over type variables.

Freshness

We also introduce the concept of variable *freshness*. When the constraint-based system introduces a type variable, there needs to be a guarantee that the variable is *fresh*—i.e., that the variable appears in no other constraints or types. If the variable is not fresh, then it's possible to inadvertently establish relationships between types that do not in fact exist. (Or worse, end up with infinite types!) We leave the details abstract and just assume we have a fresh supply of variables and we write "α fresh" when we want to assert that α was taken from this fresh supply.[11]

[9] The Damas-Milner system is commonly presented with *type schemes* that do allow quantification over type variables, but this quantification is restricted to **let**-expressions (and exists to enable polymorphism) and is not equivalent to System F's universal quantification.

[10] Or, at the very least, it's considered an error (although there are no formal checks in place).

[11] It's actually simple to iron out the details here: one approach is to simply track the used variables from the type variable set. But, these details aren't particularly interesting and make for busier notation. Another approach is to simply assume a fresh variable stack that contains no repeated entries, which is essentially how this presentation proceeds.

2.5.2 Constraints

A *constraint* is simply an equality between two types. A *constraint set* is a set of such equalitites:

$$\{\tau_i = \sigma_i \mid i \in 1 \ldots n\}$$

Substitutions

With the introduction of type variables, we naturally also introduce *type substitutions*. Type substitutions are of the same stuff as term substitutions in Section 2.1.3, except we'll use use R instead of S when referring to them and they operate on *type* variables instead of term variables. As with terms, they work as one would expect on types:

$$\begin{aligned}
R(t) &= t \\
R(\tau_1 \to \tau_2) &= R(\tau_1) \to R(\tau_2) \\
R(\tau_1, \tau_2) &= (T(\tau_1), R(\tau_2)) \\
R(\#id_1\ \tau_1 \mid \cdots \mid \#id_n\ \tau_n) &= \#id_1\ R(\tau_1) \mid \cdots \mid \#id_n\ R(\tau_n) \\
R(\alpha) &= \begin{cases} \tau & \text{if } [\alpha \mapsto \tau] \in R \text{ for some } \tau \\ \alpha & \text{otherwise} \end{cases}
\end{aligned}$$

and on typing contexts:

$$R([x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n]) = [x_1 \mapsto R(\tau_1), \ldots, x_n \mapsto R(\tau_n)]$$

We also allow type substitutions to operate on terms, substituting the types in functions and type ascriptions:

$$\begin{aligned}
R(c) &= c \\
R(x) &= x \\
R(e : \tau) &= R(e) : R(\tau) \\
R(e_1, e_2) &= (R(e_1), R(e_2)) \\
R(\lambda x : \tau. e) &= \lambda x : R(\tau). R(e) \\
R(e_1 e_2) &= R(e_1) R(e_2) \\
R(\#id\ e) &= \#id\ R(e) \\
R(\textbf{match}\ e\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n) &= \textbf{match}\ R(e)\ \textbf{case}\ p_1 \to R(e_1) \cdots \textbf{case}\ p_n \to R(e_n)
\end{aligned}$$

We say that a type substitution R *unifies* a constraint set $\{\tau_i = \sigma_i \mid i \in 1 \ldots n\}$ if $R(\tau_i) = R(\sigma_i)$ for all i. R *resolves* the constraint set if it unifies the set and each constraint $R(\tau_i) = R(\sigma_i)$ contains no type variables.

2.5.3 Typing relation

Since we're using a constraint-based DM type system, we extend our standard three-place relation to one involving constraints. A *constrained typing relation* augments the typing relation from Section 2.3.1 with a set of constraints C, written

$$\Gamma \vdash x : \tau \ \parallel\ C$$

which can be understood as the term x having type τ under environment Γ subject to the constraints in C. That is, if there is a substitution R that unifies C, then $x : R(\tau)$ under environment $R(\Gamma)$.

2.5.4 Binding pattern variables

The definition of *bind* in Section 2.3.2 won't work under the constrained typing relation: it relies on the structure of types, which may be unknown in the presence of type variables. Fortunately, the solution is simple. In our definition of *bind* for the constrained relation—which we'll call *cbind*—we just generate a fresh type variable for each term variable in a pattern and add it to the context. The typing rules will automatically generate constraints on the fresh type variables to ensure that they have the correct types. *cbind* is defined as:

$$cbind(p) = \begin{cases} \{x : \alpha\} & p = x,\ \alpha \text{ fresh} \\ cbind(p_1) \cup cbind(p_2) & p = (p_1, p_2) \\ cbind(p') & p = \#id_p\ p' \\ \varnothing & \text{otherwise} \end{cases}$$

2.5.5 Typing rules

Figure 2.4 gives the typing rules for the constraint-based system. Each rule may generate type constraints. Typing an expression using the rules yields a type and a set of constraints. A suitable unification algorithm then returns a substitution R that resolves the constraint set (or fails).

$$\frac{}{\Gamma \vdash c : const(c) \ \parallel\ \varnothing}\ \textsc{C-Const} \qquad \frac{x : \tau_1 \in \Gamma}{\Gamma \vdash x : \tau_2 \ \parallel\ \{\tau_1 = \tau_2\}}\ \textsc{C-Var}$$

$$\frac{\Gamma \vdash e : \tau_2 \ \parallel\ C}{\Gamma \vdash (e : \tau_1) : \tau_1 \ \parallel\ \{\tau_1 = \tau_2\} \cup C}\ \textsc{C-Ascrip}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \parallel\ C_1 \qquad \Gamma \vdash e_2 : \tau_2 \ \parallel\ C_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2) \ \parallel\ C_1 \cup C_2}\ \textsc{C-Tuple}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \ \parallel\ C}{\Gamma \vdash (\lambda x : \tau_1 . e) : \tau_1 \to \tau_2 \ \parallel\ C}\ \textsc{C-Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \parallel\ C_1 \qquad \Gamma \vdash e_2 : \tau_2 \ \parallel\ C_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1 e_2 : \alpha \ \parallel\ C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\}}\ \textsc{C-App}$$

$$\frac{\Gamma \vdash e : \tau \ \parallel\ C \qquad \alpha, \beta \text{ fresh}}{\Gamma \vdash (\#id\ e) : \alpha \ \parallel\ \{\alpha = \#id\ \tau \mid \beta\} \cup C}\ \textsc{C-Constr}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \ \parallel\ C \qquad \alpha \text{ fresh} \qquad C_e = \{\texttt{true} = exhaustive(\cup_{i=1}^{n}\{p_i\}, \tau)\} \\ \text{for each } i \in 1 \ldots n\text{:} \\ cbind(p_i) \vdash p_i : \tau \ \parallel\ C_i \qquad \Gamma \cup cbind(p_i) \vdash e_i : \alpha \ \parallel\ C_i' \end{array}}{\Gamma \vdash (\textbf{match}\ e\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n) : \alpha \ \parallel\ C \cup (\cup_{i=1}^{n} C_i \cup C_i') \cup C_e}\ \textsc{C-Match}$$

Figure 2.4: Constraint-based typing rules.

The C-Var rule and *cbind*

It may seem odd that C-Var wasn't defined as

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \parallel \varnothing}$$

The reason for the current definition is that, without the added $\tau_1 = \tau_2$ constraint, *cbind* would fail to establish a relationship between the generated type variables for pattern variables and the types of the variables that the variables bind to. The $cbind(p_i) \vdash p_i : \tau \parallel C_i$ premise in C-Match generates constraints that tie the types of p_i's variables to τ, where τ is the type of the expression being matched upon, thereby simplifying the definition of *cbind*.

Constructor ascriptions

A key point in typing terms involving sum types is that the structure of a sum type may only be partially known at any given time. We represent this partial knowledge via constraints[12]; the constraint

$$\{\alpha = \#id\ \tau_1 \mid \beta\} \tag{2.1}$$

from the C-Constr rule constrains α to be a sum type with the clause $\#id\ \tau_1$ and additional, currently unknown, clauses β. Note that the $\mid$ operator is both associative and commutative, hence the order in which the constraints are constructed is immaterial. Unlike in the Constr rule, a type ascription is not required in the C-Constr rule. Instead, the constraint in Equation (2.1) allows the type system to delay resolving β (and hence α) until later. Without type variables, the full sum type must be known to construct a valid typing (which is why the Constr rule requires an ascription).

The generated constructor constraints require at least one base truth to fully resolve the unknown β variable against. This means that there must exist a constraint that relates the sum type type variables to a fully resolved sum type. Such a constraint is introduced by a type ascription (including ascriptions in function binders). Even with the constraint-based typing, there must be at least one sum type ascription in the source program. But, the constraint-based system frees the programmer from writing redundant ascriptions for sum types. Section 4.4 discusses true polymorphic sum types, which allow for ascriptionless programs.

On the note of redundant ascriptions, C-Fun could have alternatively been defined as

$$\frac{\Gamma, x : \alpha \vdash e : \tau \parallel C \qquad \alpha \text{ fresh}}{\Gamma \vdash (\lambda x \,.\, e) : \alpha \rightarrow \tau \parallel C}$$

(with the addition of ascriptionless functions to our terms grammar) freeing the programmer from yet further unnecessary ascriptions, while also permitting function polymorphism (since the bound variable is no longer tied to an explicit type). We forgo this modified typing rule to simplify the proof of soundness for the constraint-based system. (We want to maintain as much similarity as possible between the monomorphic and constraint-based systems.)

[12] *Row variables* [Wan, Gar03] are commonly used for this purpose.

The exhaustivity check

In the C-MATCH rule, the *exhaustive* check from MATCH is included as a constraint instead of a premise. This notation is somewhat sloppy: strictly speaking, *exhaustive* is a predicate, and its codomain isn't (necessarily) in the set of types. (And our analysis doesn't assume the existence of booleans either.) At any rate, the necessary technical adjustments here are straightforward and few. Another possibility would be to extend the constraint-based relation to a five-place relation that included a place for a set of *exhaustive* predicates.

This change to the exhaustivity predicate is required because the type passed to *exhaustive* may not be fully resolved until later. For the same reason, the Futhark compiler performs a pattern exhaustivity check only after the type checking phase has resolved all types, see Section 3.4.

2.5.6 Soundness of the constraint-based system

Adding type ascriptions

As we did for the monomorphic type system of Section 2.3, we'd like to assert that the constraint-based system is sound. To do so, we'll show that terms that are well-typed under the constraint-based relation are also well-typed under the standard typing relation from Section 2.3.1. Unfortunately, there's a problem! Not all programs that are well-typed under the constraint-based relation are well-typed under the standard typing relation. The issue is the removal of the type ascription requirement in the C-CONSTR rule, which increases the set of programs that are well-typed under the constraint-based relation. What is necessary is a transformation that adds type ascriptions to constructors, which we call *add*:

$$
\begin{aligned}
add(\#id\ e : \tau) &= (\#id\ add(e)) : \tau \\
add(\#id\ e) &= (\#id\ add(e)) : \alpha && (\alpha \text{ fresh}) \\
add(\lambda x : \tau. e) &= \lambda x : \tau. add(e) \\
add(e : \tau) &= add(e) : \tau \\
add((e_1, e_2)) &= (add(e_1), add(e_2)) \\
add(e_1 e_2) &= (add(e_1) add(e_2)) \\
add(\textbf{match}\ e\ \textbf{case}\ p_1 \to e_1 \cdots \textbf{case}\ p_n \to e_n) &= \textbf{match}\ add(e)\ \textbf{case}\ p_1 \to add(e_1) \cdots \textbf{case}\ p_n \to add(e_n) \\
add(e) &= e
\end{aligned}
$$

Of course, we want to make sure that *add* preserves typing:

Lemma 2.12. If $\Gamma \vdash e : \tau \parallel C$ then $\Gamma \vdash add(e) : \tau \parallel C$.

Proof. By induction on $\Gamma \vdash e : \tau \parallel C$. All cases except C-CONSTR just involve invoking the inductive hypothesis (where applicable) and then applying that case's typing rule.

Case C-CONSTR:

$$e = \#id\ e' \qquad \tau = \alpha \qquad C = \{\alpha = \#id\ \tau \mid \beta\} \cup C' \qquad \Gamma \vdash e' : \tau \parallel C'$$

By the inductive hypothesis, $\Gamma \vdash add(e') : \tau \parallel C'$. By C-CONSTR,

$$\Gamma \vdash (\#id\ add(e')) : \alpha \parallel \{\alpha = \#id\ \tau \mid \beta\} \cup C'$$

By C-Ascrip,

$$\Gamma \vdash ((\#id\ add(e')) : \alpha) : \alpha \ \ \| \ \ \{\alpha = \alpha\} \cup \{\alpha = \#id\ \tau \mid \beta\} \cup C'$$

By the definition of *add*,

$$\Gamma \vdash add(\#id\ e') : \alpha \ \ \| \ \ \{\alpha = \alpha\} \cup \{\alpha = \#id\ \tau \mid \beta\} \cup C'$$

And since $\{\alpha = \alpha\}$ is trivially satisfied by any substitution,

$$\Gamma \vdash add(\#id\ e') : \alpha \ \ \| \ \ \{\alpha = \#id\ \tau \mid \beta\} \cup C'$$

which is just $\Gamma \vdash add(\#id\ e') : \alpha \ \ \| \ \ C$.

We are being sloppy with the generated fresh type variables here: there are no guarantees that the fresh variables chosen for building the $\Gamma \vdash e : \tau \ \ \| \ \ C$ relation will be the same for $\Gamma \vdash add(e) : \tau \ \ \| \ \ C$ the relation. That is, the two constraint sets, types, and typing contexts could differ up to variable names. But, since the two generated constraint sets are structurally identical (after removal of superfluous constraints like $\alpha = \alpha$), variable renaming is straightforward and we assume any necessary renaming is done behind the scenes. □

With *add* in hand, we can now show that well-typed terms under the constraint-based relation are well-typed under the standard relation, if constraints are annotated with types.

Theorem 2.13. Suppose that $\Gamma \vdash e : \tau \ \ \| \ \ C$. If R resolves C, then $R(\Gamma) \vdash R(add(e)) : R(\tau)$.

Proof. By induction on $\Gamma \vdash e : \tau \ \ \| \ \ C$.

Case C-Const:

$$e = c \qquad\qquad \tau = const(c) \qquad\qquad C = \varnothing$$

Since $R(c) = c$, $R(const(c)) = const(c)$ and $add(c) = c$, $R(\Gamma) \vdash R(add(c)) : R(const(c))$ by Const.

Case C-Var:

$$e = x \qquad\qquad \tau = \tau_2 \qquad\qquad C = \{\tau_1 = \tau_2\} \qquad\qquad x : \tau_1 \in \Gamma$$

Since $x : \tau_1 \in \Gamma$, $R(x : \tau_1) \in R(\Gamma)$. By Var, $R(\Gamma) \vdash R(add(x)) : R(\tau_1)$.

Case C-Ascrip:

$$e = e' : \tau_1 \qquad\qquad \tau = \tau_1 \qquad\qquad C = \{\tau_1 = \tau_2\} \cup C' \qquad\qquad \Gamma \vdash e' : \tau_2 \ \ \| \ \ C'$$

By the inductive hypothesis, if R' resolves C', then $R'(\Gamma) \vdash R'(add(e')) : R'(\tau_2)$. Since $C' \subset C$, any resolver for C must also resolve C'. Hence, $R(\Gamma) \vdash R(add(e')) : R(\tau_2)$ and, because $R(\tau_1) = R(\tau_2)$, we have $R(\Gamma) \vdash R(add(e')) : R(\tau_1)$. By Ascrip and the definition of *add*, $R(\Gamma) \vdash R(add(e' : \tau_1)) : R(\tau_1)$.

Case C-Tuple:

$$e = (e_1, e_2) \qquad\qquad \tau = (\tau_1, \tau_2) \qquad\qquad C = C_1 \cup C_2 \qquad\qquad \Gamma \vdash e_1 : \tau_1 \ \ \| \ \ C_1$$

$$\Gamma \vdash e_2 : \tau_2 \ \ \| \ \ C_2$$

Since $C = C_1 \cup C_2$, if R resolves C then R must also resolve C_1 and C_2. By the inductive hypothesis, $R(\Gamma) \vdash R(add(e_1)) : R(\tau_1)$ and $R(\Gamma) \vdash R(add(e_2)) : R(\tau_2)$. By Tuple, $R(\Gamma) \vdash R(add((e_1, e_2)) : R(\tau_1, \tau_2)$.

Case C-Fun:

$$e = \lambda x : \tau_1 . e \qquad \tau = \tau_1 \rightarrow \tau_2 \qquad \Gamma, x : \tau_1 \vdash e : \tau_2 \ \| \ C$$

By the inductive hypothesis, $R(\Gamma, x : \tau_1) \vdash R(add(e)) : R(\tau_2)$. By Fun, $R(\Gamma) \vdash \lambda x : R(\tau_1). R(add(e)) : R(\tau_1) \rightarrow R(\tau_2)$ which can be rewritten as $R(\Gamma) \vdash R(add(\lambda x : \tau_1 . e)) : R(\tau_1 \rightarrow \tau_2)$.

Case C-App:

$$e = e_1 e_2 \qquad \tau = \tau_3 \qquad C = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow \tau_3\} \qquad \Gamma \vdash e_1 : \tau_1 \ \| \ C_1$$

$$\Gamma \vdash e_2 : \tau_2 \ \| \ C_2$$

By the inductive hypothesis (and the fact that R must also resolve C_1 and C_2 separately), $R(\Gamma) \vdash R(add(e_1)) : R(\tau_1)$ and $R(\Gamma) \vdash R(add(e_2)) : R(\tau_2)$. Additionally, $R(\tau_1) = R(\tau_2) \rightarrow R(\tau_3)$, so $R(\Gamma) \vdash R(add(e_1)) : R(\tau_2) \rightarrow R(\tau_3)$. By App, $R(\Gamma) \vdash R(add(e_1 e_2)) : R(\tau_3)$.

Case C-Constr:

$$e = \#id\ e' \qquad \tau = \alpha \qquad C = \{\alpha = \#id\ \tau' \mid \beta\} \cup C \qquad \Gamma \vdash e : \tau \ \| \ C$$

By the inductive hypothesis, $R(\Gamma) \vdash R(add(e')) : R(\tau')$. Since R resolves C, $R(\alpha) = R(\#id\ \tau' \mid \beta) = \#id\ R(\tau') \mid R(\beta)$. Since $R(\alpha)$ is a sum type with a $\#id\ R(\tau')$ clause, by Constr, $R(\Gamma) \vdash (\#id\ add(e') : R(\alpha)) : R(\alpha)$. We can pull R outside of the constructor term: $R(\Gamma) \vdash R(\#id\ add(e') : \alpha) : R(\alpha)$ and by the definition of add have $R(\Gamma) \vdash R(add(\#id\ e')) : R(\alpha)$.

Case C-Match:

$$e = \textbf{match}\ e'\ \textbf{case}\ p_1 \rightarrow e_1 \cdots \textbf{case}\ p_n \rightarrow e_n \qquad \tau = \alpha$$

$$C_e = \{\texttt{true} = exhaustive(\cup_{i=1}^{n}\{p_i\}, \tau')\} \qquad C = C \cup (\cup_{i=1}^{n} C_i \cup C_i') \cup C_e$$

$$\Gamma \vdash e' : \tau' \ \| \ C$$

for each $i \in 1 \ldots n$: $\qquad cbind(p_i) \vdash p_i : \tau' \ \| \ C_i \qquad \Gamma \cup cbind(p_i) \vdash e_i : \alpha \ \| \ C_i'$

The inductive hypothesis gives $R(\Gamma) \vdash add(e') : R(\tau')$, $R(cbind(p_i)) \vdash R(add(p_i)) : R(\tau')$, and $R(\Gamma \cup cbind(p_i)) \vdash R(add(e_i)) : R(\alpha)$. By C-Match,

$$R(\Gamma) \vdash R(add((\textbf{match}\ e'\ \textbf{case}\ p_1 \rightarrow e_1 \cdots \textbf{case}\ p_n \rightarrow e_n)) : R(\alpha)$$

(Note that $R(add(p_i)) = p_i$, since patterns don't have type ascriptions.)

□

Theorem 2.14 (Soundness)**.** If $\vdash e_1 : \tau \ \| \ C$, C has a resolver R and $R(add(e_1)) \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_n$, then e_n is a value or there exists e_{n+1} such that $add(e_n) \rightsquigarrow add(e_{n+1})$.

Proof. By Lemma 2.12, $\vdash add(e) : \tau \ \| \ C$. By Theorem 2.13, $\vdash R(add(e)) : R(\tau)$. By Theorem 2.11, we're done. □

2.5.7 Unification

The typing rules in Figure 2.4 are only half of the type inference story: the generated constraint set still needs to be solved in order to resolve the type variables in a typing relation. Solving this constraint set is called *unification*. In the constraint-based system, *syntactic* unification, as first described in [Rob65], is generally sufficient to build a unifying substitution. There is one issue: sum type constraints involving the $|$ operator cannot be solved by a syntactic unifier. The issue lies in the associativity and commutativity of $|$. For example, the constraint

$$\#a\ \tau_a \mid \alpha = \#b\ \tau_b \mid \beta$$

cannot be solved syntactically because it's impossible to make the left side match the right side in a syntactical sense, even if the types are equal due to the algebraic properties of $|$.[13] Our out here is to allow equality modulo some algebraic property; unifiers of such constraint sets are called *equational* unifiers [BS01]. We construct a set E of algebraic identities that $|$ obeys:

$$E = \{\#a\ \tau_a \mid \#b\ \tau_b = \#b\ \tau_b \mid \#a\ \tau_a, \quad (\#a\ \tau_a \mid \#b\ \tau_b) \mid \#c\ \tau_c = \#a\ \tau_a \mid (\#b\ \tau_b \mid \#c\ \tau_c)\}$$

and then modify our constraint to be equal *modulo E* (meaning, up to the identities in the set E), which we denote with $=_E$:

$$\#a\ \tau_a \mid \alpha =_E \#b\ \tau_b \mid \beta \tag{2.2}$$

By relaxing our constraints in Figure 2.4 to be equal module E instead of simply equal, we obtain a constraint set that can be solved by an equational unification algorithm.

2.5.8 A note on polymorphism

A clarification is in order: while the DM type system is often used polymorphically (via let-polymorphism), the current type system is a monomorphic system, aside from the constructor polymorphism introduced by the C-Constr rule in Figure 2.4.

The C-Constr rule means that constructors are typed polymorphically; however, as previously discussed, the only way to fully resolve a sum type (i.e., the α type variable appearing in the C-Constr rule) is via a type ascription. This means that while constructors are polymorphic, expressions that operate on sum types (i.e., functions and matches) end up being monomorphic in regards to sum types because the type ascription concretely determines the type these expressions operate on. Constructors are typed polymorphically as a necessary consequence of generating useful constraints in the type reconstruction process.

[13] It is interesting to note that symbolic unification works fine on types constructed with the function type operator $\rightarrow$. The reason for this is that the $\rightarrow$ operator is neither associative nor commutative, so in the unification of the constraint $\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4$, τ_1 and τ_3 may be structurally compared. This luxury is unavailable for constraints of the form $\#id_1\ \tau_1|\alpha_1 = \#id_2\ \tau_2|\alpha_2$, where τ_1 and τ_2 may only be compared if $id_1 = id_2$ (otherwise, the constructor corresponding to $\#id_1$ may be somewhere in α_2, if at all).

Chapter 3

Implementation

3.1 Overview

This chapter covers the implementation of sum types in the Futhark compiler, which is written in the Haskell[14] programming language. The implementation consists of three main parts:

1. Extending Futhark's syntax
2. Type checking
3. Conversion to Futhark's intermediate representation (internalization)

In the subsequent sections, we'll examine each of these areas of the implementation.

3.2 Syntax

3.2.1 A few differences and a few specifics

Futhark's sum type syntax largely mirrors that of the small language introduced in Section 2.1, but features a number of significant differences. First, sum types and constructors are generalized to be k-ary instead of strictly 1-ary:[15]

$$\begin{array}{rcll} \tau & ::= & \cdots & \\ & \mid & \#id_1\ \tau_1^1\ \cdots\ \tau_{k_1}^1 \mid \cdots \mid \#id_n\ \tau_1^n\ \cdots\ \tau_{k_n}^n & \text{sum type} \\ & & & \\ e & ::= & \cdots & \\ & \mid & \#id\ e_1\ \cdots\ e_k & \text{constructor} \end{array}$$

Under this specification, 0-ary constructors are permitted, i.e., enumerations. Futhark also features *parametric type abbreviations* which enables the programmer to give alternative names to types and, via type application, fill-in any type parameters. For example,

$$option\ \alpha = \#some\ \alpha \mid \#none$$

[14]https://www.haskell.org/

[15]For continuity, we continue using the notation developed in Chapter 2. Technically, τ now refers to Futhark's type grammar and not the grammar in Section 2.1.1. For didactic reasons, the presentation is only an approximation of the actual Futhark language. See https://futhark.readthedocs.io/en/latest/language-reference.html for an up-to-date full specification.

defines the type abbreviation *option* which accepts one type parameter; e.g., *option* τ is equivalent to #*some* τ | #*none*. **Match**-expressions and constructors are syntactically identical to the theoretical treatment.

As in the theory chapter, recursive types are disallowed as well and—also as in the theory chapter—Futhark's type system is entirely *structurally* typed. This means that constructors exist in the global namespace. All constructors are always in scope. As a consequence, unlike many functional programming languages with sum types, partial application of constructors is forbidden in Futhark: a k-ary constructor must be initialized with exactly k arguments. (Getting partial application right in a structural type system is no easy task!)

3.2.2 Parsing and internal representation

Futhark's parser is generated via the Happy[16] parser generator. Happy produces a parser by reading a grammar specification of the language; adding support for sum types amounted to extending the grammar in a straightforward fashion and defining the relevant abstract syntax tree (AST) extensions in the Futhark compiler.

Sum types in the AST

In this section we'll take a look at how the Futhark compiler represents sum types. The Haskell snippets in this section are approximation only—they've been simplified for didactic purposes. Figure 3.1 below shows the (simplified) primary modifications to Futhark's AST types.

```
data Type     = ... | Sum (Map Name [Type])

data Exp      = ... | Constr Name [Exp] Type
                    | Match Exp [(Pattern, Exp)] Type

data Pattern =        TuplePattern [Pattern]
                    | RecordPattern [(Name, Pattern)]
                    | PatternParens Pattern
                    | Id Pattern
                    | Wildcard Pattern
                    | PatternAscription Pattern Type
                    | PatternLit Exp Type
                    | PatternConstr Name Type [Pattern]
```

Figure 3.1: Sum type additions to Futhark's AST types. The `Name` type is approximate to the set of labels $\mathcal{L}$ in Section 2.1.1. Only the `PatternLit` and `PatternConstr` constructors were added as part of adding sum type support to Futhark; the remaining `Pattern` constructors were already present in the code base.

Sum types are internally represented as a mapping between constructor labels and a list of types.[17] This is a natural fit for sum types: each clause is disjoint from the

[16] https://www.haskell.org/happy/

[17] This list of types cannot be arbitrarily long; while Section 3.2.1 described Futhark as supporting k-array sum types, in reality this is only true for $k \leq 256$ due to the tagging mechanism; see Section 2.3 for details.

rest and the map enables convenient look-up for the typing checking phase of the compiler.

Prior to the addition of sum types, Futhark exclusively supported *irrefutable* patterns; i.e., patterns that cannot fail to match a value and are represented by the first six pattern constructors in Figure 3.1. (These patterns would appear in **let**-expressions and functions.) The addition of **match**-expressions and constructors expanded the set of applicable patterns to include the `PatternLit` constructor (when we want to match on an exact value) as well as the `PatternConstr` (constructors whose fields are replaced with patterns, similar to Section 2.1.4). These new patterns (or patterns featuring these patterns as sub-patterns) *can* fail to match a value. This is because both of these new pattern constructors involve fixed values that only some values of a certain type may match with. To address pattern failures, a pattern exhaustivity checker was added to Futhark (Section 3.4).

3.3 Type checking

3.3.1 Constraints and unification

Futhark uses a constraint-based Damas-Milner type system, similar to that of Section 2.5. Unlike in Section 2.5, there is no hard distinction between type inference/constraint-generation and unification: constraints are solved on an ad-hoc basis rather than being accumulated all at once.

The basis of Futhark's constraints is its `Constraint` type, which expresses a certain constraint on a given type variable. A number of constructors from this type are shown below (to demonstrate the variety of possible constraints), including the `HasConstrs` constraint that was added to support type checking sum types.

```
data Constraint = ... | Constraint Type
                      | Equality
                      | HasConstrs (Map Name [Type])
```

As an example, if a type variable has a `Constraint t` constraint, the type variable is constrained to have type `t` (i.e., `t` may be substituted in place of the type variable). The `Equality` constraint constrains a type variable to be a type that supports equality. The new `HasConstrs m` constraint approximates the equational constraint in Equation (2.2) and enforces that a type variable is a sum type and that it has at least the clauses described by the map `m`.

Sum type unification between two types `t1` and `t2` can be split into three scenarios, depending on the types being unified (i.e., explicit sum types or type variables) and, in the case of type variables, the constraints on them. Note that before two types are unified, any existing type substitutions (including those specified by a `Constraint` constraint) are applied to both types.

1. (`t1` is a type variable and `t2` is a sum type): If `t1` has a `HasConstrs` constraint, all clauses in the constraint must be present in the sum type. If so, `t1` is mapped to `t2`, and `t2` and substituted for `t1` in the existing constraint set. If `t1` has other constraints that conflict with unification with a sum type, unification fails and the type checker signals an error. If `t1` has no other constraints, it's simply mapped to `t2` without checking for clause presence in `t2`.

2. (`t1` and `t2` are type variables): In the unification of two distinct type variables both constrained by a `HasConstrs` constraint, the constraint is updated to include the intersection of the two constraint sets and one type variable is mapped to the other. In the case of incompatible constraints, unification fails.

3. (`t1` and `t2` are sum types): Two explicit sum types are unified by equality, up to clause re-ordering.

3.3.2 Another note on polymorphism

The comments in Section 2.5.8 apply equally well to Futhark. Namely, the polymorphic nature of the `HasConstrs` constraint means that while constructors can be used polymorphically in Futhark, any sum type acceptors (e.g., matches and functions) are monomorphic in regard to sum types. Additionally, as in the theoretical development, any program that includes constructors must feature a type ascription at some point. Without such an ascription, there is no sum type that type variables with a `HasConstrs` constraint can resolve against.

This restriction can be lifted somewhat in the case of **match**-expressions: in lieu of a type ascription, the compiler can construct a sum type from the constructor patterns being matched on. For example, from the match

```
match x
case #a 5      -> e1
case #b (1, 2) -> e2
case #c false  -> e3
case #d        -> e4
```

the compiler could infer that

```
x : #a i32 | #b (i32, i32) | #c bool | #d
```

by simply merging the `HasConstrs` constraints of each constructor pattern into a sum type. This sort of inference is safe: if the programmer forgets a constructor in the match expression, that constructor will pop-up elsewhere in the program and a type error will be signaled by the compiler. Futhark does not currently support this sort of sum type inference, but could with only minimal modification to its type system.

3.4 Pattern exhaustivity

Futhark disallows inexhaustive pattern matches in **match**-expressions and performs an exhaustivity check after type checking (type information must be known to check pattern exhaustivity) to enforce exhaustive matches. Before sum types, Futhark only supported irrefutable patterns where a successful match is guaranteed by the type system and is not dependent upon values. In the presence of literal and constructor patterns, we now must exhaustively check the cases of a **match**-expression for complete coverage of the value space of the type being matched on.

The exhaustive check involves splitting the patterns of a match's cases into vertical columns and ensuring that each column is exhaustive. For constructors, this involves ensuring that there is a corresponding case for each clause of the sum type. For literal values, either all possibilities must be present (e.g., `true` and `false` for booleans) or a catch-all pattern must be present (e.g., for numeric types). If patterns consist

of sub-patterns, the sub-patterns themselves are also checked for exhaustivity. To illustrate, we'll run through the action of the exhaustivity checker on the example **match**-expression in Figure 3.2.

```
type option 'a  = #some a | #none
type mytype = #a (i32, (i32, i32)) (option i32) | #b i32 i32

match (#b 1 2 : mytype)
case #a (1, (2, 3)) (#some 1) -> ...
case #a (4, (5, x)) (#some 2) -> ...
case #a (_, (6, _)) (#some _) -> ...
case #b x 1 -> ...
case #b _ 2 -> ...
```

Figure 3.2: Illustration of exhaustive pattern matching in the compiler. Each box represents a "column" of patterns being checked for exhaustivity. Nested boxes illustrate calls of the checker on sub-patterns. `i32` is a 32-bit integer type.

The exhaustivity checker is run left-to-right on each vertical column of patterns in the match's cases, shown by the blue boxes in the figure. Field patterns are grouped by constructor: patterns belonging to distinct constructors are separated. This is why the patterns matching on the first field of the `#a` constructor are boxed separately from the patterns matching the first field of the `#b` constructor, despite being in the same vertical column.

First, the constructors in the first blue box are checked for exhaustivity. Since the match is on a value of type `mytype`, the compiler checks that there is at least one row with an `#a` constructor and at least one row with a `#b` constructor. There is, so the check moves to the next column to the right. Before the checker checks the next column, it records the current *pattern context*: in this case, either `(#a _ _)` or `(#b _ _)`. As the moves left-to-right (and descends into sub-patterns), it builds a pattern context in order to report meaningful errors to the user.

After checking the constructor column, the checker moves right to the blue box surrounding field patterns corresponding to the `#a` constructor's first field of type `(i32, (i32, i32))`. Here, the checker descends into the sub-patterns which make up the pair pattern. In one of the sub-patterns, the checker detects a missing catch-all pattern (since `2, 5` and `6` cannot match all integers) and outputs the the pattern context `(#a (_, (p, _)) _)` to the programmer with an error. See Figure 3.3 below for the complete output generated by the checker on the example match.

```
(#a (_, (p, _)) _) where p is not one of [2, 5, 6]
(#a _ #none )
(#b _ p) where p is not one of [1, 2]
```

Figure 3.3: The missing pattern matches reported for the match in Figure 3.2.

Upon detection of a missing pattern, the checker does not check further sub-patterns of the current pattern (as resolution of the current missing pattern may make

all sub-patterns trivially exhaustive), but it does continue in its left-to-right traversal and checks for further missing patterns. As the checker continues in the fashion described it generates two further errors, as shown in Figure 3.3.

The exhaustivity checker described in this section is closely related to the pattern match compiler described by Sestoft [Ses96]. Indeed—as Sestoft shows—checking patterns for exhaustivity and compiling matches can be accomplished with the exact same algorithm. Section 4.3.1 describes a scheme to transform Futhark's matches in a method closely resembling the exhaustivity checker in this section, further demonstrating the connection between pattern compilation and pattern exhaustivity checking.

Eliminating redundant cases

The exhaustivity machinery may be modified in a straightforward manner to also report redundant cases in matches (this is not yet implemented in the compiler). For each column checked, the checker can add any rows with overlapping entries (e.g., the column $\begin{bmatrix} x \\ 4 \\ 4 \end{bmatrix}$ contains multiple overlapping entries: since x matches anything, it overlaps with both 4 entries, and the 4 entries overlap with each other) to an overlapping set and remove rows from the set which no longer overlap given the current column. After checking is complete, any rows that remain in the overlapping set must be redundant.

3.5 Internalization

This phase of the compiler involves transforming source programs written in Futhark to an intermediate representation (IR) called the *core language*. A number of compiler passes are performed on the parsed AST in preparation for IR transformation, including a *defunctorization* pass for transforming away modules, a *monomorphization* pass, and a *defunctionalization* [HHE18] pass for Futhark's higher-order functions.

3.5.1 The monomorphization pass

The first compiler pass we're concerned with is monomorphization. Monomorphization replaces polymorphic function calls with an equivalent monomorphic function, with any type variables in the polymorphic function replaced by concrete types.[18] Adding **match**-expression support here is straightforward: the existing transformation logic is simply called on the sub-expressions of the **match**-expression.

Constructor removal

Constructor expressions are simply a way to "package" existing data. Tuples act much the same in this regard[19] with the exception that constructor expressions are *tagged* in the form of an identifier. However, we can emulate the identifier tag in a tuple by simply setting the first element of the tuple to a numeric representation of the

[18]The monomorphization pass also does some other housekeeping, including expanding record variables into record patterns and removal of unreachable functions.

[19]At least when partial application isn't allowed!

identifier, given by a function $f : \mathcal{L} \to \mathbb{Z}$. This gives rise to the following constructor-to-tuple transformation

$$\#id\ e_1\ \cdots\ e_n \Longrightarrow (f(id), e_1, \ldots, e_n)$$

In practice, the function f is quite easy to define: simply sort the constructors of a sum type (by sorting the labels of their constructors) and assign each a number based on its sorted position—given a constructor $\#id$, $f(id)$ returns its assigned number. For example, assuming a lexicographic-based sorting function *sort* and a type $option\ \tau = \#some\ \tau \mid \#none$ for a type τ,

$$\begin{aligned} sort(option) &= \#none \mid \#some\ \tau \\ f(none) &= 0 \\ f(some) &= 1 \end{aligned}$$

This scheme works fine for Futhark because constructors are monomorphic after type inference has filled in all type information, so all constructors for a given type are known. In languages with polymorphic sum types, defining a function f isn't quite as simple because not all constructors may be known—a hash function with a reasonably low collision rate does the trick [Gar98].[20]

A problem with the transformation

Compare the transformation of $\#some\ x$ and $\#none$:

$$\begin{aligned} \#some\ x &\Longrightarrow (f(some), x) \\ \#none &\Longrightarrow (f(none)) \end{aligned}$$

Notice that the two transformations differ structurally. Transforming constructors with the transformation as-is yields ill-typed expressions (e.g., when matching on a constructor). In order to ensure structural similarity, each constructor is instead translated into a tuple of tuples which has fields for *every* constructor of the sum type. All such fields must be populated by values, so fields of other constructors are simply filled with default values synthesized by a function $d : \mathcal{T} \to \mathcal{E}$, which returns some default value for any type $\tau \in \mathcal{T}$. Such a function is simple to define: the actual value itself is immaterial, all that matters is that some value (of the correct type) exists as a placeholder. For example, a reasonable default value for a numeric type is simply 0. Using d we can define a function d_c which constructs default place-holder tuple representing these fields: [21]

$$d_c(\#id_i\ \tau_1\ \cdots\ \tau_{k_i}) = \begin{cases} () & k_i = 0 \\ d(\tau_1) & k_i = 1 \\ (d(\tau_1), \ldots, d(\tau_{k_i})) & \text{otherwise} \end{cases}$$

The transformation of $\#id_i\ e_1\ \cdots\ e_{k_i}$ with sorted type $\#id_1\ \tau_1^1\ \cdots\ \tau_{k_1}^1 \mid \cdots \mid \#id_n\ \tau_1^n\ \cdots\ \tau_{k_n}^n$ now becomes

$$\#id_i\ e_1\ \cdots\ e_{k_i} \Longrightarrow \Big(f(id_i), d_c(\#id_1\ \tau_1^1\ \cdots\ \tau_{k_1}^1), \ldots, (e_1, \ldots, e_{k_i}), \ldots, d_c(\#id_n\ \tau_1^n\ \cdots\ \tau_{k_n}^n)\Big)$$

[20] A collision is easily detectable by the compiler, which could either correct the collision by internally search-and-replacing the constructor name, or simply signal an error to the user.

[21] In practice, no entry is needed for 0-ary constructors. Setting the entry to () just simplifies the presentation.

With this modified transformation, our example transformation becomes (recalling that both constructors have type *option* τ)

$$\#some\ x \Longrightarrow (1, (), x)$$
$$\#none \Longrightarrow (0, (), d(\tau))$$

We perform a very similar transformation for constructor patterns, the difference being that in lieu of default values, we replace fields for other constructors with wildcard patterns. This is reassuring—default values can't matter if they're simply ignored by patterns they're matched against!

The constructor transformation is desirable because a) it simplifies the core language of the compiler and—with the appropriate **match**-expression transformation—is semantically equivalent and b) it's quite simple to implement. Unfortunately, the transformation is rather inefficient: the larger the sum type, the greater the overhead, regardless of the size of the constructor being translated. In Section 4.2, we'll discuss a modification that lets us trim down the transformation.

Since the monomorphization pass is the first post type checking pass that the compiler does on expressions, it's an opportune time to apply the constructor-to-tuple transformation. (Otherwise, we have to add constructor support to other passes!) After the transformation, the resulting tuple expression is monomorphized via existing transformation logic in the compiler.

3.5.2 Translation to Futhark's core language

While Futhark's core language could directly support constructors and matches (making the discussed transformations unnecessary), this would be at the cost of a significant increase in complexity of the compiler. Out of a desire to keep things simple, Futhark's core language was not modified as part of this project: matches are instead transformed into an equivalent core language representation.

The basic scheme is to transform **match**-expressions into a nested series of **if-else**-expressions whose boolean conditions test whether a case pattern matches the expression being matched upon and whose bodies correspond to the body of a given case. For a **match**-expression of the form

$$\begin{aligned}
&\textbf{match}\ e \\
&\textbf{case}\ p_1 \rightarrow e_1 \\
&\vdots \\
&\textbf{case}\ p_n \rightarrow e_n
\end{aligned}$$

we generate an **if-else**-expression of the form

$$\begin{aligned}
&\textbf{if}\ cond(p_1, e)\ \textbf{then}\ pbind(p_1, e, e_1)\ \textbf{else}\ (\\
&\vdots \qquad\qquad\qquad \vdots \\
&\textbf{if}\ cond(p_{n-2}, e)\ \textbf{then}\ pbind(p_{n-2}, e, e_{n-2})\ \textbf{else}\ (\\
&\textbf{if}\ cond(p_{n-1}, e)\ \textbf{then}\ pbind(p_{n-1}, e, e_{n-1})\ \textbf{else}\ pbind(p_n, e, e_n)) \cdots)
\end{aligned}$$

where *cond* is a function that takes a pattern and an expression and generates an equation that holds if and only if p successfully matches e. *pbind* is analogous in

spirit to the *bind* and *cbind* from Chapter 2: $pbind(p_i, e, e_i)$ evaluates e_i under an environment augmented with the bindings produced by matching the pattern p_i against e. Note that $pbind(p_i, e, e_i)$ is guaranteed to be well-defined as a product of the well-typedness of the given **match**-expression. All (sub-)patterns that produce bindings are irrefutable, so *pbind* make sense regardless of the particular value being matched on.

Due to the exhaustivity check described in Section 3.4, no conditions need be generated for the final case expression **case** $p_n \to e_n$: if all other conditions fail, then this case *must* succeed as a consequence of the exhaustivity of the match. Indeed, it is impossible for matches to get stuck if they've made it this far in the compiler pipeline.[22]

Generating conditions

We define *cond* in terms of a helper function *cond′* which takes a pattern p and returns a holed list of conditions, where the holes are placeholders for the expression on which p is being matched. To demonstrate, we define *cond′* for the pattern grammar given in Section 2.1.4:[23]

$$cond'(p) = \begin{cases} \lambda e.\,[e = c] & p = c \\ \lambda e.\,[] & p = x \\ \lambda e.\,(cond'(p_1))(fst(e)) +\!\!+ (cond'(p_2))(snd(e)) & p = (p_1, p_2) \\ \lambda e.\ \bot & p = \#id\ p_1\ \cdots\ p_k \end{cases}$$

where $\bot$ indicates a failure condition (no constructor patterns should be present in the program after the monomorphization pass). *cond* may then be defined in terms of a standard fold:

$$cond(p, e) = fold\left(\texttt{\&\&}, \texttt{true}, (cond'(p))e\right)$$

where `&&` is the standard boolean *and* operator, `true` the base conditional, and $(cond'(p))e$ the list being folded over (the direction of the fold is immaterial). The essence of *cond* is to enforce equality constraints between pattern constants and the expression being matched by building up the appropriate projection machinery on the value being matched on for the given pattern constant.

[22] We hope so, at least.

[23] We haven't fully defined the notation here, but it has its usual meanings. Namely, $[e_1, e_2, \dots]$ is a list and $+\!\!+$ is the list concatenation operator. Also, $fst(e_1, e_2) = e_1$ and $snd(e_1, e_2) = e_2$. We're also using k-ary constructor patterns instead of the 1-arr constructors in Section 2.1.4.

Chapter 4

Evaluation and improvements

In this chapter, we discuss a few deficiencies of the existing implementation and offer possible improvements.

4.1 Problematic arrays of sum types

As an array language, naturally Futhark supports arrays, written $[e_1, e_2, \ldots, e_n]$. The type $[]\tau$ describes an array with elements of type τ. Futhark also allows programmers to annotate array types with optional *size annotations* [Hen17]. Size annotations are written as $[n]\tau$, indicating an array of type τ with n elements and are checked dynamically instead of statically by the type checker.

Now, returning to the *option* type described in Section 3.5.1, consider the transformation on a value of type $option\ []int$ (where int is an example integer type):

$$\#some\ [1,2] \Longrightarrow (1, (), [1,2])$$

The problem at hand arises when we have an array of $option\ []int$ values:

$$[\#none,\ \#sum[1,2]] \Longrightarrow [(0, (), []), (1, (), [1,2])]$$

(Where the default value for $[]int$—used as a placeholder in the transformation of the $\#none$ constructor—is just the empty array $[]$, i.e., $d([]int) = []$). All arrays in Futhark must be regular [Hen17]. This means that there must exist an m such that

$$[(0, (), []), (1, (), [1,2])] : [n](int, (), [m]int)$$

Now, m must simultaneously be 0 and 2. Hence, the above expression is illegal and results in a run-time failure. If the default value function had access to a size annotation, it could yield a compatible default value. For example, $d([n]int) = [d(int), \cdots, d(int)]$ where the array has n elements. Unfortunately, size annotations aren't part of the type checker and hence there is no mechanism in place to annotate $\#none$ with the type $option\ [2]int$ instead of $option\ []int$. Fortunately, work is being actively done on adding size annotation support to the type checker, at which point they'll mature to *size types* [Hen19b]. Once this work is complete, this issue can be addressed.

4.2 Constructor deduplication

The constructor transformation discussed in Section 3.5.1 is rather inefficient: each constructor for a given sum type is represented by a tuple with entries for *every* field

of the sum type. To do better, [Hen19a] describes and implements a method to avoid some duplication by having distinct constructors share tuple entries.[24]

The transformation is moved out of the monomorphization pass and into the internalization phase, where terms and types are converted to Futhark's core language. For each clause of a sum type, the transformation constructs a mapping which maps the fields of the clause to indices in the output tuple. The key idea is that for each primitive type (numeric types or booleans), the resulting tuple needs only have as many entries for that type as the clause with the maximal number of fields with the type in question. Such a mapping can be constructed in a straightforward manner by iterating over each clause of a sum type: if the tuple being generated already has a free entry with the appropriate type for a field of the clause, the mapping assigns the field to that entry, rather than appending a new entry to the tuple. The resulting mapping is best illustrated by a picture—see Figure 4.1 for an example.

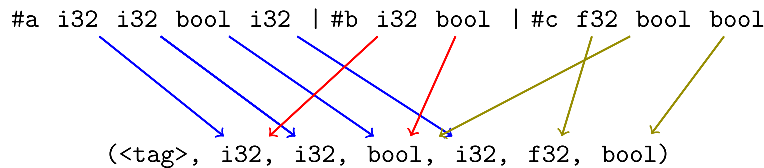

Figure 4.1: Illustration of the constructor transformation with deduplication. The arrows map fields of constructors to entries in the tuple representation for the sum type. Notice that a number of arrows map to the same entry—this is how the transformation saves space!

For large sum types or sum types whose individual clauses share fields with many of the same types, the transformation can result in a significant performance increase: [Hen19a] reports a factor of two speed-up in a ray tracer compared with the transformation of Section 3.5.1.

4.3 Efficiency of generated conditional statements

4.3.1 Eliminating redundant tests

The **match**-expression transformation described in Section 3.5.2 unnecessarily generates **if**-**else**-expressions that test values more than once. The issue is that—for *each* pattern—the scheme described generates a condition that tests *all* values required for a match. For example, the expression

$$\begin{array}{l} \textbf{match}\ e \\ \textbf{case}\ (0,5,3) \to e_1 \\ \textbf{case}\ (0,x,3) \to e_2 \\ \textbf{case}\ (0,4_) \to e_3 \\ \textbf{case}\ _ \to e_4 \end{array}$$

[24] This improved constructor transformation has already been added to the Futhark compiler by Troels Henriksen.

is transformed into

$$
\begin{array}{l}
\textbf{if } ((e.1 == 0)\ \&\&\ (e.2 == 5)\ \&\&\ (e.3 == 3)) \\
\quad \textbf{then } \mathit{pbind}((0,5,3), e, e_1) \\
\quad \textbf{else } (\textbf{if } ((e.1 == 0)\ \&\&\ (e.3 == 3)) \\
\quad\quad \textbf{then } \mathit{pbind}((0,x,3), e, e_2) \\
\quad\quad \textbf{else } (\textbf{if } ((e.1 == 0)\ \&\&\ (e.2 == 4)) \\
\quad\quad\quad \textbf{then } \mathit{pbind}((0,4,_), e, e_3) \\
\quad\quad\quad \textbf{else } \mathit{pbind}(_, e, e_4)))
\end{array}
$$

where $e.k$ returns the k-th entry of the tuple e. This scheme results in significant back tracking: in the above, $(e.1 == 0)$ may be tested up to three times and $(e.3 == 3)$ up to two times. To avoid this, [Car84] describes an algorithm which produces *decision trees* via splitting patterns vertically into columns in a process remarkably similar to how pattern exhaustivity is checked in Section 3.4. Reasonable decision trees of this nature never test a value more than once. To illustrate, the patterns in the example program above are split into three columns:

$$
\begin{array}{l}
\textbf{case } (0, 5, 3) \to e_1 \\
\textbf{case } (0, x, 3) \to e_2 \\
\textbf{case } (0, 4, _) \to e_3 \\
\textbf{case } (_, _, _) \to e_4
\end{array}
$$

(Where _ is expanded into (_, _, _) to structurally match the other patterns. Such an expansion is always possible and doesn't change the semantics of the match.) We are now tasked with choosing a column to begin discriminating on. The goal is to produce as small of an expression as possible. A number heuristics exist to do so: [Car84] suggests choosing the column with the largest number of literal values while [Mar08] suggests a more sophisticated heuristic system that combines multiple metrics. For simplicity, we proceed left-to-right across the columns.[25] In the first column we must test whether the tuple entry is 0 or not, yielding the expression

$$
\begin{array}{l}
\textbf{if } (e.1 == 0) \\
\textbf{then } (\cdots) \\
\textbf{else } (\cdots)
\end{array}
$$

In the second column we group further discriminating expressions by the entries in the first column. When the first entry is 0, we must discriminate on whether the second entry is 5, 4, or a variable/wildcard. When the first entry isn't 0, the second

[25] In practice, [SR00] suggests that proceeding left-to-right is actually perfectly reasonable and that heuristics really only make a performance difference in pathological cases. [SR00] also notes that it's possible to compile any match into an ideal expression, but that this minimization process is likely NP-complete.

entry is another wildcard, so no further discrimination is required. This yields

$$
\begin{aligned}
&\textbf{if } (e.1 == 0)\\
&\textbf{then } (\textbf{if } (e.2 == 5)\\
&\quad \textbf{then } (\cdots)\\
&\quad \textbf{else } (\textbf{if } (e.2 == 4)\\
&\quad\quad \textbf{then } (\cdots)\\
&\quad\quad \textbf{else } (\cdots)))\\
&\textbf{else } (\cdots)
\end{aligned}
$$

A similar process for the third column yields

$$
\begin{aligned}
&\textbf{if } (e.1 == 0)\\
&\textbf{then } (\textbf{if } (e.2 == 5)\\
&\quad \textbf{then } (\textbf{if } (e.3 == 3)\\
&\quad\quad \textbf{then } \mathit{pbind}((0,5,3), e, e_1)\\
&\quad\quad \textbf{else } (\cdots))\\
&\quad \textbf{else } (\textbf{if } (e.2 == 4)\\
&\quad\quad \textbf{then } (\textbf{if } (e.3 == 3)\\
&\quad\quad\quad \textbf{then } \mathit{pbind}((0,x,3), e, e_2)\\
&\quad\quad\quad \textbf{else } \mathit{pbind}((0,4,_), e, e_3)))\\
&\quad\quad \textbf{else } (\textbf{if } (e.3 == 3)\\
&\quad\quad\quad \textbf{then } \mathit{pbind}((0,x,3), e, e_2)\\
&\quad\quad\quad \textbf{else } \mathit{pbind}((_,_,_), e, e_4)))\\
&\textbf{else } \mathit{pbind}((_,_,_), e, e_4)
\end{aligned}
$$

We've completed traversing the columns, but there remains a superfluous **else**$(\cdots)$ line with no expressions to fill it in the expression. Such lines only occur when there is no alternative expression to return, meaning we may combine them with the previous discriminator:

$$
\begin{aligned}
&\textbf{if } (e.1 == 0)\\
&\textbf{then } (\textbf{if } ((e.2 == 5)\ \&\&\ (e.3 == 3))\\
&\quad \textbf{then } \mathit{pbind}((0,5,3), e, e_1)\\
&\quad \textbf{else } (\textbf{if } (e.2 == 4)\\
&\quad\quad \textbf{then } (\textbf{if } (e.3 == 3)\\
&\quad\quad\quad \textbf{then } \mathit{pbind}((0,x,3), e, e_2)\\
&\quad\quad\quad \textbf{else } \mathit{pbind}((0,4,_), e, e_3)))\\
&\quad\quad \textbf{else } (\textbf{if } (e.3 == 3)\\
&\quad\quad\quad \textbf{then } \mathit{pbind}((0,x,3), e, e_2)\\
&\quad\quad\quad \textbf{else } \mathit{pbind}((_,_,_), e, e_4)))\\
&\textbf{else } \mathit{pbind}((_,_,_), e, e_4)
\end{aligned}
$$

Even when combining tests like this, each test is still performed at most once as long as the && operator supports short-circuiting (which, in Futhark, it does). While this

alternative scheme does produce a larger expression (that may be exponential in size, compared to the linear size of the naive matching previously described), it executes faster due to a smaller number of comparisons. In practice, matches that programmers write tend to be simple and code explosion is a non-issue.

As noted in Section 3.4, the Futhark compiler also does not eliminate redundant—perhaps “unreachable” is the better term here—cases. These cases remain unreachable in the transformed **if**-**else**-expression and so don’t appreciably add to the runtime of compiled programs, but they do contribute to increased code size. Further efficiency gains can be expected by implementing the redundant elimination algorithm detailed in Section 3.4 and at virtually no additional cost: the algorithm can be coupled together with the exhaustivity checker (and performed in the same pass).

4.4 Sum type polymorphism

As discussed in Section 3.3, Futhark only supports very limited sum type polymorphism in the form of polymorphic constructors—a necessary consequence of supporting good type inference for sum types. In this section, we’ll explore the necessary adjustments required to support full sum type polymorphism through the lens of the *OCaml*[26] programming language.

OCaml supports both “standard” nominally typed sum types (which it calls *variants*) and so-called *polymorphic variants* (i.e., polymorphic sum types). As with Futhark, OCaml’s polymorphic variants are structurally typed and constructors exist in the global namespace. The difference lies in that OCaml’s polymorphic variants are just that: polymorphic—both for constructors *and* acceptors. To enable true polymorphism, OCaml uses a more sophisticated sum type constraint system. To illustrate, consider the ill-typed Futhark program below:

```
let f x =
 match x
 case #a -> 1
 case #b -> 2
 case #c -> 3
```

When compiled by the Futhark compiler, the following type error is returned:

```
Type is ambiguous (must be a sum type with constructors: #a | #b | #c).
```

Here, the compiler is pretty-printing the generated `HasConstrs` constraint (see Section 3.3.1) as the error message. Adopting the notation of [Gar98], we may alternatively express this constraint as

$$x : [> \#a \mid \#b \mid \#c]$$

which expresses that `x` must be a sum type with *at least* the given constructors. The $>$ symbol is suggestive here: the ascription is a sort of lower bound on the type of `x`.[27] One might ask if it makes sense to have a corresponding constraint

$$x : [< \#a \mid \#b \mid \#c]$$

[26] https://ocaml.org/

[27] Just to be clear: the above is a *constraint*, not a type. There is actually a hidden type variable, which we can make explicit: $x : \forall[\alpha > \#a \mid \#b \mid \#c]\ .\ \alpha$. This constraint is analogous to the $\alpha = \beta \mid \#a \mid \#b \mid \#c$ constraint from Section 2.5.

that, instead of a lower bound, act as an *upper bound*, placing a maximal restriction on the clauses of a sum type. Looking at the ill-typed Futhark program above, *this* actually seems like the correct constraint to ascribe to the program: the match includes a case for each of the constructors in the constraint, so data with a type that is upper bounded by this constraint should be safe to match upon.

This is the approach that OCaml takes: by using both upper and lower bounds, OCaml's polymorphic variants are truly polymorphic in that upper-bounded constraints allow the type system to express constraints on acceptors without needing to fully resolve their precise types. In this system, expressions may be "typed" with constraints and then interact with types that meet these constraints.

Modifying Futhark's type system to support sum type polymorphism in this way is entirely possible (namely by the introduction of an upper bound sum type constraint). However, it's less clear how to modify Futhark's back-end to support polymorphic sum types where the lack of static type information significantly complicates transformation to Futhark's core language. The interesting point here is that—at the type level—supporting sum type polymorphism when sum types are structurally typed and constructors exist in the global namespace amounts to little more than an extra constraint.

Finally, polymorphic sum types aren't without disadvantages: the OCaml manual[28] specifically notes a weaker type discipline and loss of static type information, which can be consequential for highly-optimizing compilers. Benefits—aside from doing away with type ascriptions (which can be reduced following the remarks in Section 3.3.2 anyway)—mostly involve things like code-reuse as well as a sort of type-level simulation of sub-typing, neither of which are particularly relevant for Futhark [Gar00, Gar98].

4.5 The utility of sum types in Futhark

In this section, we'll take a look at a couple of real-world Futhark programs with and without sum types.

Diving Beet

Diving Beet is a cellular-automaton based particle simulator, written in Futhark (and Python) by Troels Henriksen.[29] The simulator encodes various *elements* as numeric values; a few token examples are reproduced below:

```
type element = u8

let oil : element = 6u8
let water : element = 7u8
let fire : element = 12u8
let fire_end : element = 22u8
```

Notably, the fire element spans a range of values between `fire` and `fire_end` in order to simulate fire burning out over time (e.g., `17` represents a fire element that is halfway to burning out). This information can be represented better and safer via sum types. The range of fire elements is combined into a single constructor with an age payload:

[28] http://caml.inria.fr/pub/docs/manual-ocaml/lablexamples.html#sec49

[29] https://github.com/athas/diving-beet

```
type age = u8
type element = #oil | #water | #fire age
```

Lys

Lys[30] is an SDL[31]-based graphics programming library written in Futhark. The Futhark program defines a `lys` module type which features a number of event handlers for different possible input events (e.g., a key press or mouse movement). The intent is that the user of the library defines a module of type `lys` which implements the desired behavior. Prior to the introduction of sum types, the module types for the possible input event handlers were[32]

```
type key_event = #keydown | #keyup

val key : key_event -> i32 -> state -> state
val mouse : (mouse_state: i32) -> (x: i32) -> (y: i32) -> state -> state
val wheel : (x: i32) -> (y: i32) -> state -> state
```

With sum types, all these events were combined into a single `event` type and the three handlers were reduced to one:

```
type event = #step f32
             | #keydown {key:i32}
             | #keyup {key:i32}
             | #mouse {buttons:i32, x:i32, y:i32}
             | #wheel {x:i32, y:i32}

val event : event -> state -> state
```

(A `#step` event is just a time step event.) Aside from the module type definition being conceptually simpler and more cohesive, this change also results in a nicer definition of the `event` handler when a module of type `lys` is defined. Rather than having to define a handler for each event separately (as would have been required pre sum types), instead the handler is a single function:

```
let event (e: event) (s: state) =
 match e
    case (#step td) -> ...
    case (#wheel {x, y}) -> ...
    case (#mouse {buttons, x, y}) -> ...
    case _ -> s
```

Notably, the match on the event type lets the programmer easily ignore events that they do not care about (here, for key presses). In the sum type version, the library can also be easily updated to support additional events: if another input type is desired (e.g., from a joystick), simply add an appropriate `#joystick` constructor to the `event` type. This may not even break user programs if they have a catch-all case in their definition of the `event` function.

[30] https://github.com/diku-dk/lys

[31] https://www.libsdl.org/

[32] Observant readers will note that `key_event` is a sum type. This part of the library was written when Futhark supported the special case of 0-ary sum types (i.e., enumerations) as part of the implementation process of *k*-ary sum types.

Chapter 5

Conclusions and future work

We have described the theory and implementation of sum types in Futhark. On the theory side, we explored a type system of a model language with sum types and proved its soundness. The constraint-based type system discussed in Section 2.5 is a convincing facsimile of Futhark's actual type system in regards to sum types: the results of this section inspire confidence in the correctness of the corresponding implementation in the Futhark compiler.

While we're confident in the correctness of the type system there are still improvements to be made. As Futhark's type system becomes increasingly complex with the introduction of new features like sum types and—in the future—size types (see Section 4.1) type error messages become an increasingly vital form of feedback for the programmer. Futhark's structural type system makes this a difficult task: structural type systems lack the granularity of nominal type systems (as they can only distinguish types by structure alone). This shortcoming is especially apparent with the introduction of sum types. For example, in a nominal type system the constructor *#none* from the *option* τ type in Section 3.5.1 can only be of type *option* τ. In contrast, Futhark's structural system can make no such distinction: *#none* is a clause of an unbounded number of sum types simultaneously, which decreases the amount of information that is available as feedback to the programmer and has negative ramifications for error locality. Solutions to this problem remain to be investigated; a *bidirectional* type system (see, e.g., [DK19]) has been suggested as a possible solution .

The internalization part of the implementation leaves a bit more to be desired. While the implementation described in this work is certainly functional (aside from the issue described in Section 4.1) and was thoroughly tested with nearly 100 unit tests, it isn't particularly efficient! Even with the efficiency improvements described in Chapter 4 there remains work to be done in this area, especially in the transformation of constructors into tuples—even with deduplication (Section 4.2), tuples are still larger than they need be. (Deduplication only works for fields of the same type, even if that field could fit values of different types. That is, a more efficient implementation would allow a single tuple entry to perhaps hold both booleans and numeric values.)

Qualitative evaluation of sum types (Section 4.5) in Futhark shows that sum types do indeed have a place in specialized computational languages and are useful to programmers. Collections of distinct values arises in all sorts of computational problems: providing the programmer a type-level interface for such values—backed by a static type checker—has proven to be a useful abstraction that not only leads to more expressive programs, but safer ones too.

Bibliography

[Aug85] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[Ble95] Guy E Blelloch. *NESL: A nested data-parallel language*. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, 1995.

[BS01] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[Car84] Luca Cardelli. Compiling a functional language. *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP 84*, 1984. doi:10.1145/800055.802037.

[CDDK86] Dominique Cl6ment, Jolle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-lvil. 1986.

[CKL+11] Manuel M.t. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. Mcdonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. *Proceedings of the sixth workshop on Declarative aspects of multicore programming - DAMP 11*, 2011. doi:10.1145/1926354.1926358.

[CLPJ+07] Manuel Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. pages 10–18, 01 2007. doi:10.1145/1248648.1248652.

[CPN16] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *ACM SIGPLAN Notices*, volume 51, pages 378–391. ACM, 2016.

[DJK+18] Derek Dreyer, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, and David Swasey. Semantics of type systems, 02 2018. Lecture Notes.

[DK19] Joshua Dunfield and Neel Krishnaswami. Bidirectional typing, 2019. arXiv:1908.05839.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 82*, 1982. doi:10.1145/582153.582176.

[EHAO18] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large. *Proceedings of the ACM on Programming Languages*, 2(ICFP):97:1–97:30, July 2018.

[Gar98] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 13, page 7. Baltimore, 1998.

[Gar00] Jacques Garrigue. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*, 2000.

[Gar03] Jacques Garrigue. Simple type inference for structural polymorphism. 01 2003.

[Hen17] Troels Henriksen. *Design and implementation of the Futhark programming language*. PhD thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, 2017.

[Hen19a] Troels Henriksen. Futhark 0.12.1 released, Aug 2019. URL: https://futhark-lang.org/blog/2019-08-21-futhark-0.12.1-released.html.

[Hen19b] Troels Henriksen. Towards size types in futhark, Aug 2019. URL: https://futhark-lang.org/blog/2019-08-03-towards-size-types.html#.

[HHE18] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming*, TFP 2018. Springer-Verlag, September 2018.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1238844.1238856, doi:10.1145/1238844.1238856.

[HSE+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3062341.3062354, doi:10.1145/3062341.3062354.

[HTEO19] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP 2019, pages 53–67, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3293883.3295707, doi:10.1145/3293883.3295707.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1411304.1411311, doi:10.1145/1411304.1411311.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[PR05] François Pottier and Didier Remy. *The Essence of ML Type Inference*. MIT Press, 2005.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan 1965. doi:10.1145/321250.321253.

[Ses96] Peter Sestoft. Ml pattern match compilation and partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, pages 446–464, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[SR00] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical report, Charlottesville, VA, USA, 2000.

[Wan] Mitchell Wand. Corrigendum: Complete type interference for simple objects. *[1988] Proceedings. Third Annual Information Symposium on Logic in Computer Science*. doi:10.1109/lics.1988.5111.