Polymorphic CSP Type Checking

Ping Gao and Robert Esser
Concurrent and Real-time Systems Laboratory
Department of Computer Science
Adelaide University
Adelaide 5005, Australia
{ping, esser}@cs.adelaide.edu.au

Abstract

Communicating Sequential Processes (CSP) is a language used to describe and reason about concurrent systems. It consists of a process algebra combined with a functional language. This combination poses unique problems when attempting to design a type checker. In this paper the differences between a conventional functional language type checker and a type checker for the CSP language are discussed. This type checker was developed to identify polymorphic types, an important first step towards the goal of automating data independence [4]. The result of this work has been incorporated into the Adelaide Refinement Checker [8] (ARC) - a CSP based tool suite for model checking concurrent systems.

1 Introduction

As originally proposed by Hoare, CSP [2] is a process algebra used to model and reason about concurrent systems. Subsequently it has developed into a rich language [13, 12] better able to describe realistic systems with complex data communication. This has been achieved by augmenting CSP with a functional expression language [9], similar to Haskell [1], while preserving the underlying communication model and synchronization mechanism.

The combination of process algebra and a functional expression language has an impact not only on the CSP syntax but also on its semantics. From a semantic point of view, the process algebra sits above the expression language while syntactically the process algebra has been incorporated into the expression language. The whole language can be viewed as an applied λ-calculus [4].

Similar to ML-like functional programming languages [10], a CSP program hardly mentions types at all1.

1The only exception being the definition of channels.

In this style of programming users can define polymorphic functions – functions that work on objects of varying types. Often the absence of a type checker for CSP leads to subtle type errors only being found at run-time. As CSP is typically used for model checking, these errors often occur after significant computation – a waste of computer resources and a frustration for the user.

In addition to discovering type inconsistencies, type checking also provides information important for reasoning about data independence. Data independence [4] is a technique aimed at reducing the large (or infinite) state space required to represent large (or infinite) types in model checking. Data independence identifies types that have little or no influence on the result of model checking. For such types, Lazic [4] has shown that reducing the size of these types will not influence the result of model checking. This has the effect that the size of the state space is often dramatically reduced – improving performance in terms of time and memory. A type checker that identifies polymorphic types is an important first step towards the goal of automating data independence.

The CSP type checker is based on Milner’s type inference algorithm [6] and Robinson’s unification algorithm [11] with extensions made to handle features specific to the CSP language. In this paper additional type constructors are introduced together with type constraints due to the hybrid nature of CSP. Specifically the unification of qualified types (types with constraints) and types formed by new type constructors is described. Finally a type inference algorithm for CSP that preserves polymorphism is presented.

2 The Classical Type Inference Algorithm

Before describing the CSP type inference algorithm it is necessary to describe how types are inferred using a classical polymorphic type inference algorithm [6]. This is best illustrated by considering the following example of a generic unbounded buffer.
The process `Buff` has three parameters `in`, `out` and `s` representing respectively the input channel, output channel and the sequence in which elements are stored. The fact that the input and output channels are parameters of `Buff` implies that the buffer is able to handle all communicable values. Typically `Buff` is invoked with an empty sequence `s`. Hence the function `# s`, returning the number of elements in sequence `s`, will return 0 and the only action `Buff` can execute is to input a value from the channel `in` and to invoke the `Buff` process with a sequence containing `x`, i.e. `Buff (in, out, s')`. With a non-empty sequence the `else` branch of the `if` statement will execute. Here the process either inputs another value from the `in` channel and appends this to the sequence or outputs the head element `head(s)` of the sequence and invokes the `Buff` process with the head element of the sequence removed, i.e. `Buff (in, out, tail(s))`.

The process `Buff` has three parameters `in`, `out` and `s` representing respectively the input channel, output channel and the sequence in which elements are stored. The fact that the input and output channels are parameters of `Buff` implies that the buffer is able to handle all communicable values. Typically `Buff` is invoked with an empty sequence `s`. Hence the function `# s`, returning the number of elements in sequence `s`, will return 0 and the only action `Buff` can execute is to input a value from the channel `in` and to invoke the `Buff` process with a sequence containing `x`, i.e. `Buff (in, out, s')`. With a non-empty sequence the `else` branch of the `if` statement will execute. Here the process either inputs another value from the `in` channel and appends this to the sequence or outputs the head element `head(s)` of the sequence and invokes the `Buff` process with the head element of the sequence removed, i.e. `Buff (in, out, tail(s))`.

The types of the polymorphic sequence functions are:

$$
\sigma_\# : \langle \alpha \rangle \rightarrow \text{int}
$$

$$
\sigma_{\text{head}} : \langle \alpha \rangle \rightarrow \alpha
$$

$$
\sigma_{\text{tail}} : \langle \alpha \rangle \rightarrow \langle \alpha \rangle
$$

$$
\sigma_s : \langle \alpha \rangle \rightarrow \langle \alpha \rangle
$$

where $\sigma_{\text{exp}}$ represents the type of expression $\text{exp}$ and $\alpha$ a generic type variable.

Using the approach in [6], for each occurrence of a polymorphic operation, $\alpha$ will be substituted by a new type variable referred to as a substitution instance — denoted by $\beta_1, \beta_2, \cdots$. The structure of expressions imposes constraints upon their types. Hence from the sequence functions and the `Buff` process the following constraint equations are obtained where the left hand side of $\rightarrow$ is used to indicate the expression and the right hand side the type constraint.

$$
\# s \rightarrow \sigma_s = \langle \beta_1 \rangle
$$

$$
\text{head}(s) \rightarrow \sigma_s = \langle \beta_2 \rangle
$$

$$
\text{tail}(s) \rightarrow \sigma_s = \langle \beta_3 \rangle
$$

$$
in\ x \rightarrow \sigma_{\text{in}} = \sigma_s \Rightarrow \text{event}
$$

$$
\exists \ x \rightarrow \sigma_s = \langle \sigma_x \rangle
$$

$$
\text{out} \cdot \text{head}(s) \rightarrow \sigma_{\text{out}} = \langle \beta_2 \Rightarrow \text{event} \rangle
$$

$$
\text{Buff} (\text{in}, \text{out}, \langle \alpha \rangle) \rightarrow \sigma_{\text{Buff}} = \sigma_{\text{in}} \Rightarrow \sigma_{\text{out}} \Rightarrow \langle \sigma_x \rangle \rightarrow \text{process}
$$

Also of note is that new types have been introduced to represent values in the process algebra part of the language. These types are taken from an extended set of types containing both types from the expression language and the process algebra. The new types used in the above example, such as `process`, `event`, `sequence type` $\langle \tau \rangle$ and yield type $\tau_1 \Rightarrow \tau_2$ will be explained in section 4.

The above equations can be solved by applying Robinson's unification algorithm [11] extended to deal with the new process algebra types. Hence the type of the `Buff` process becomes:

$$
(\beta_1 \Rightarrow \text{event}) \Rightarrow (\beta_1 \Rightarrow \text{event}) \Rightarrow \langle \beta_1 \rangle \rightarrow \text{process}
$$

As $\beta_1$ is a free substitution instance it can be substituted by the generic type variable $\alpha$. Hence the type of `Buff` is:

$$
(\alpha \Rightarrow \text{event}) \Rightarrow (\alpha \Rightarrow \text{event}) \Rightarrow \langle \alpha \rangle \rightarrow \text{process}
$$

Using the classical approach to type checking implies that the protocol type of the channels `in` and `out` is totally general. However in CSP, functions, processes or sets may not be communicated along channels. This is due to the absence of equality tests for these types necessary for event synchronization. To type check CSP a means of constraining the generic type variable $\alpha$ is needed so that any undesirable instantiations can be rejected. In addition to the equality test constraint imposed by event synchronization, other constraints are introduced by the language. These will be discussed in section 4.2 and their effect on the unification algorithm in section 5.

The classical approach also has difficulty with the oft encountered multi-part communication. Here CSP uses the `dot` type (see section 4) to represent the type of the multi-part data communicated along channels.

### 3 The CSP Language

In this section a subset of the CSP language is defined. As mentioned in section 1 there are two views of the language: A semantic point of view where the process algebra sits above the the expression language and the expression language plays a supportive role and a syntactical point of view where the process algebra is an integral part of the expression language providing facilities for building processes.

The syntax for a subset of CSP is defined as:
\[ \text{e ::= c} \]
\[ x \]
\[ \lambda x \cdot e \]
\[ \mu x \cdot e \]
\[ e_1 e_2 \]
\[ \text{let } x = e_1 \text{ within } e_2 \]
\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]
\[ (e_1, \ldots, e_n) \]
\[ \{e_1, \ldots, e_n \mid gs\} \]
\[ e_1, \ldots, e_n \]
\[ \{e_1, \ldots, e_n \mid gs\} \]
\[ \text{event set} \]
\[ ch f_1 \cdots f_n \rightarrow P \]
\[ e_1 \sqcap e_2 \]
\[ e_1 \sqcap e_2 \]
\[ e_1 \setminus e_2 \]
\[ e[[R]] \]
\[ e_1 || e_2 \]
\[ e_1 || e_3 || e_2 \]

where gs is a list of generators used in a comprehension and each generator is either a data map \( e \leftarrow e' \) or a boolean filter \( e \leftarrow e' \) indicates that \( e \) ranges over \( e' \).

The action part of a prefixing is formed by a channel \( ch \) followed by a number of fields \( f_1, \ldots, f_n \), each being either \(?x \) (input a value and store it in \( x \)) or !e (output a value of expression \( e \)).

An event set has a sugared form \( \{e_1, \ldots, e_n\} \) representing the set of all events associated with channel \( e_1, \ldots, e_n \), e.g. with channel \( c : \text{Bool} \) the event set \( \{c \mid \} \) represents \( \{\text{true}, \text{false}\} \).

The renaming relation \( R \) specifies how an event is substituted by another and is formed by a list of event substitutions, each taking the form of \( e \leftarrow e' \) with every event associated with \( e \) replaced by the one associated with \( e' \).

### 4 The Types

In this section CSP types are classified as either being unqualified or qualified, i.e. whether they are subject to any constraints.

#### 4.1 Unqualified Types

In addition to types usually associated with functional languages, CSP includes a number of new types:

- \( \text{Process} \) and \( \text{event} \) represent the two basic objects introduced by inclusion of the process algebra.

- \( \text{Set} \) and \( \text{sequence} \) types: A sequence is an ordered collection of objects of the same sort, typically used to model a store (e.g. the buffer in section 2). The sequence type constructor \( (\) \) is similar to \textit{list} in [9]. Unlike the sequence type, elements in a set are unordered and distinct. In CSP the equality test and ordered comparison (based on sequence prefix) are defined for sequences but not for sets.

- \( \text{Union} \) types are expressed as the sum of types. Each alternative consists of a data tag \( (l_i \mid 1 \leq i \leq m) \) followed by a list of types derived directly from the type definitions in a CSP script.

- \( \text{Dot} \) types: The dot type represents a list of values separated by dots. Dot types represent the protocol of a channel and the type of the field prefixed by the data tag. The degenerate case, consisting of a singleton value, indicates either an atomic channel (having no data communication) or a data tag of \( 0 \)-arity.

- \( \text{Yield} \) types: A yield type \( \tau_1 \Rightarrow \tau_2 \) represents the object that when supplied with a value of type \( \tau_1 \) yields a value of type \( \tau_2 \). In CSP the yield type represents channels\(^2\) and data tags. For channels \( \tau_1 \) is the channel protocol type and \( \tau_2 \) the \textit{event} type and for data tags, \( \tau_1 \) is the type of the field prefixed by the data tag and \( \tau_2 \) the union type. The yield type distinguishes channels and data tags from function objects.

With TVARS representing the set of all type variables, TYPES the set of all CSP types, and TAG the set of names used in datatype definitions, unqualified types \( \tau \) are defined as:

\(^2\)\textit{Channel} refers only to the channel identifier, the \textit{label}, and does not include the channel protocol type.
4.2 Qualified Types

Qualified types are types subject to additional constraints imposed by the combination of the expression language and the process algebra and the presence of overloaded operators – operators defined for a number of types e.g. the equality (\(=\)) and ordering (\(<, \leq, >, \geq\)) operators.

There are four constraints:

- The **Eq** \(\tau\) constraint requires that an equality operation for type \(\tau\) be defined. It is introduced either by an explicit equality or inequality operation or implicitly by an event synchronization. As a consequence all communicable value types must satisfy **Eq** \(\tau\).

- The **Ord** \(\tau\) constraint requires that an ordering comparison is defined for type \(\tau\). It is introduced by the \(\{>, <, \geq, \leq\}\) operations.

- The **Func** \(\tau\) constraint ensures that \(\tau\) is a well-formed function type. It ensures that a function type does not return a non-process type from a process type, e.g. \(\text{process } \rightarrow \text{int}\) is illegal.

- The **Proc** \(\tau\) constraint ensures that type \(\tau\) must be a process type. It is introduced when the argument of a function type is of type process, e.g. \(\text{process } \rightarrow \alpha\) requires that **Proc** \(\alpha\) be satisfied.

A constraint involving type \(\tau\), can immediately be resolved if \(\tau\) is fully determined, i.e., it does not contain type variables. The criterion used for checking constraints is:

- **Eq** \(\tau\) holds if \(\tau\) is neither a function, process, set type, nor any type involving these types.

- **Ord** \(\tau\) holds only if \(\tau\) is the integer type or a sequence type.

- **Func** \(\tau_1 \rightarrow \tau_2\) holds if \(\tau_1\) is a non-process type or \(\tau_2\) is a process type.

- **Proc** \(\tau\) holds if \(\tau\) is process or any composite type that contains a process type.

For a type with unresolved type variables, a constraint can be reduced according to the above criterion until only type variables remain.

In the following discussion types involved in unification and inference are qualified by the condition (type predicate) under which they are well-formed. The type predicate \(p\) is a conjunction of constraints. A type \(\tau\) qualified by such a predicate \(p\) is called a qualified type \(\phi\) and written as \(p \triangleright \tau\). Where \(p\) is trivially true, \(p \triangleright \tau\) is abbreviated to \(\tau\). In the remainder of this paper the term type will only be used where no ambiguity exists.

4.3 The Relationship Between Two Types

As in the standard approach a substitution \(S\) is defined as a map from type variables to types. A substitution applied to a qualified type \(\phi\) can be composed with other substitutions with \((S_2 \circ S_1)\phi = S_2(S_1 \phi)\).

**Definition 1** A type \(p_1 \triangleright \tau_1\) is said to be more restrictive than a type \(p_2 \triangleright \tau_2\), denoted by \((p_1 \triangleright \tau_1) \preceq (p_2 \triangleright \tau_2)\) iff for any substitution \(S\) such that \(S\tau_1 = S\tau_2\), \(S\tau_1\) implies \(S\tau_2\), i.e. if \(S\tau_1\) holds, then so does \(S\tau_2\).

**Definition 2** A type \(p_1 \triangleright \tau_1\) conflicts with a type \(p_2 \triangleright \tau_2\) iff for any substitution \(S\) such that \(S\tau_1 = S\tau_2\), if \(S\tau_1\) holds then \(S\tau_2\) fails and vice versa.

An attempt to unify two conflicting types gives rise to a type error.

5 Unification

Given two types \(\tau_1\) and \(\tau_2\), the unification algorithm determines whether the types can be unified, and if so, constructs the most general unifier (mgu) [14] \(S\) such that \(S\tau_1 = S\tau_2\).

In this section, an extension of Robinson’s unification algorithm, denoted by \(\mathcal{U}\), is presented. The extension is necessary to handle the dot and qualified types.

5.1 Unification of Dot Types

Of all the introduced unqualified types required for CSP only the dot type poses a challenge to the unification algorithm. For example consider the equality test \(1.x == 1.y\), the resulting unification is \(\mathcal{U}(\text{int}.\alpha_x, \alpha_y, \text{int})\). It is tempting to treat the unification as for other constructed types.
and break it into two smaller unifications \( \mathcal{U}(\text{int}, \alpha_y) \) and \( \mathcal{U}(\alpha_x, \text{int}) \) producing a unifier \([\alpha_x/\text{int}, \alpha_y/\text{int}]\).

Unfortunately this is not correct as it will refuse valid instantiations of \( x \) and \( y \) such as \( \text{true}/1 \) and \( \text{true}/1 \) respectively. The problem is that the above unifier is too restrictive because in particular circumstances the type variable occurring in dot types is able to be stretched to match an appropriate number of dot-joined types, e.g. \( \mathcal{U}(\text{int}, \alpha, \text{int. bool}) \) produces the unifier \([x/\text{int}, y/\text{int}]\).

As a result the following rules apply when unifying two dot types: \( \tau_1 = \tau_{11}, \ldots, \tau_{1m} \) and \( \tau_2 = \tau_{21}, \ldots, \tau_{2n}(m, n > 1) \). These rules are applied recursively in the order (R1 \( \cdots \) R7).

R1: If \( \tau_{11}, \ldots, \tau_{1m} \) is fully determined and

[R1-1] \( n = m \):

\[ \mathcal{U}(\tau_{11}, \tau_{21}), \ldots, \mathcal{U}(\tau_{1m}, \tau_{2n}) \]

[R1-2] \( n < m \) and \( \tau_{2n} = \alpha \):

\[ \mathcal{U}(\tau_{11}, \tau_{21}), \ldots, \mathcal{U}(\tau_{1(n \ 1)}, \tau_{2(n \ 1)}) \]

\[ \mathcal{U}(\tau_{1n}, \ldots, \tau_{1m}, \alpha) \]

[R1-3] otherwise: unification fails.

The symmetrical case for \( \tau_2 \) is omitted.

R2: If \( \tau_{11} = \tau_{21} \): the unification is reduced to \( \mathcal{U}(\tau_{12}, \ldots, \tau_{1m}, \tau_{22}, \ldots, \tau_{2n}) \).

R3: If \( \tau_{1m} = \tau_{2n} \): the unification is reduced to \( \mathcal{U}(\tau_{11}, \ldots, \tau_{1(m \ 1)}, \tau_{21}, \ldots, \tau_{(n \ 1)}) \).

R4: If neither of \( \tau_{11} \) and \( \tau_{21} \) is a type variable, the unification is reduced to \( \mathcal{U}(\tau_{11}, \tau_{21}), \mathcal{U}(\tau_{12}, \ldots, \tau_{1m}, \tau_{22}, \ldots, \tau_{2n}) \).

R5: If neither of \( \tau_{1m} \) and \( \tau_{2n} \) is a type variable, the unification is reduced to \( \mathcal{U}(\tau_{11}, \ldots, \tau_{1(m \ 1)}, \tau_{22}, \ldots, \tau_{2n}) \).

R6: If \( \tau_{11} \) is type variable \( \alpha \) but \( \tau_{21} \) is not:

\[ \text{add } [\alpha/(\tau_{21}, \beta)] \text{ to the substitution } S \]

\[ \mathcal{U}(\tau_{21}, \beta, \ldots, \tau_{1m}, \tau_{21}, \ldots, \tau_{2n}) \]

The symmetrical case is omitted.

R7: If \( \tau_{1m} \) type variable \( \alpha \) but \( \tau_{2n} \) is not:

\[ \text{add } [\alpha/(\beta, \tau_{2n})] \text{ to the substitution } S \]

\[ \mathcal{U}(\tau_{11}, \ldots, \beta, \tau_{2n}, \tau_{21}, \ldots, \tau_{2n}) \]

The symmetrical case is omitted.

When two dot types have type variables at both ends, e.g. \( \alpha_1, \alpha_2 \) and \( \alpha_3, \alpha_4 \) and if \( \alpha_1 \neq \alpha_3 \) and \( \alpha_2 \neq \alpha_4 \), then no type rule can be applied and a unification error results.

The idea behind R1 is that if at least one type is fully determined, e.g. \( \tau_1 \), a pair-wise unification is conducted until a type variable is encountered representing the last component of the partially determined type \( \tau_2 \). The type variable stretches from where it was encountered up to the end of \( \tau_1 \), mapping the type dot-joined by all intermediate types.

Of note are rules R6 and R7 in which the most general substitution, \([\alpha/\tau_{21}, \beta] \) or \([\alpha/\beta, \tau_{2n}] \), is created for \( \alpha \) by introducing an auxiliary type variable \( \beta \).

In the example \((1.x == y.1.)\) described at the start of section 5.1, the mgu \( U = [x/\beta, \text{int}, \alpha_5/\text{int.} \beta] \) obtained by applying rules R6 and R2.

5.2 Unification of Qualified Types

When considering qualified types, care must be taken when applying a substitution \( S \) to a type predicate \( \alpha \). If a type variable \( \alpha \) in \( \alpha \) is mapped to \( \alpha' \), it \( \tau \) in \( S \), then,

**Definition 3** \( \mathcal{U}([\alpha/\alpha'], \tau) = p[[\alpha/\tau] \wedge p'] \)

As previously mentioned the unification algorithm \( \mathcal{U} \) is an extension of Robinson’s algorithm [11] taking into account the effect of the qualified type predicate \( p \). The unification of unqualified types is therefore treated as the special case where \( p = \text{true} \). The result of the algorithm is the unified type \( \phi \) and the mgu \( S \).

Algorithm \( \mathcal{U} \)

\[ \mathcal{U}(p_1 \triangleright \tau_1, p_2 \triangleright \tau_2) = (\phi, S) \]

(1) \( \tau_1, S = \mathcal{U}(\tau_1, \tau_2) \)

(2) \( p_1' = S p_1, p_2' = S p_2 \)

(3) check \( p_1' \) and \( p_2' \)

\[ \begin{align*}
(3.1) & \text{ if either } p_1' \text{ or } p_2' \text{ fails, or they conflict with each other, then a type error is flagged and unification terminates.} \\
(3.2) & \text{ if } p_1' \text{ implies } p_2' \text{, } \phi = p_1' \triangleright \tau; \\
(3.3) & \text{ if } p_2' \text{ implies } p_1' \text{, } \phi = p_2' \triangleright \tau; \\
(3.4) & \text{ otherwise, } \phi = (p_1 \wedge p_2') \triangleright \tau; 
\end{align*} \]

The influence of the type predicates can be seen when an attempt to unify \( \text{Eq } \alpha \triangleright \alpha \) and \( \text{Proc } \alpha \triangleright \alpha \) is made as this will cause a type error. Also it can be seen that the unification of \( \text{Eq } \alpha \triangleright \alpha \) and \( \text{Ord } \alpha \triangleright \alpha \) produces the unified type \( \text{Ord } \alpha \triangleright \alpha \).

6 Type Inference

With aid of a unification algorithm, a type inference algorithm checks type compatibility and also determines the principle (most general) types [5] of arbitrary expressions.

The type inference algorithm presented in this section extends Milner’s polymorphic type checking algorithm [6]. The extension is necessary to handle CSP specific expressions, in particular dot composite expressions used to form structured data or events.

6.1 Typing the Dot Composite

An expression is usually typed in a compositional fashion, that is, once the types of its sub-expressions are determined, then the type of the expression is immediately
obtained without any extra computation. Unfortunately in CSP, dot composite expressions cannot be treated this way. A dot composite expression is a list of values separated by ’.’. For example, consider the dot composite expression $A_1.A_2.1.true$ where $A_1$ and $A_2$ are tags from the following datatype definitions:

$$\text{datatype } T_1 = A_1, T_2$$
$$\text{datatype } T_2 = A_2.\text{Int}$$

Here it can be seen that it is not possible to determine whether the expression is well-typed from the types of its subexpressions alone. The types of the subexpressions are:

$$A_1 :: T_2 \Rightarrow T_1$$
$$A_2 :: \text{int} \Rightarrow T_2$$
$$1 :: \text{int}$$
$$\text{true} :: \text{bool}$$

However, the structure of each value can be extracted from its type. Hence by analyzing component types the type of the overall expression can be deduced. This is achieved via the procedure $\text{pack}$, defined as:

$$\text{pack}(\phi_1, \ldots, \phi_n) = \text{pack}_2((\cdots (\text{pack}_2(\phi_1, \phi_2)) \cdots), \phi_n)$$

where $\text{pack}_2(\phi_1, \phi_2) = \phi$ and

(1) $\phi = \phi_1 \cdot \cdots \cdot \phi_{2n}, \phi_1 \cdot \cdots \cdot \phi_{1m} \Rightarrow \phi_1'$
   If $\phi_1 = (\phi_1 \cdots \phi_{1m} \Rightarrow \phi_1')$
   $\phi_2 = \phi_2 \cdot \cdots \cdot \phi_{2n}$ and $\phi_2' \leq \phi_1$

(2) $\phi = \phi_1 \cdot \cdots \cdot \phi_{1m} \Rightarrow \phi_1'$
   If $\phi_1 = (\tau_1 \cdot \cdots \cdot \phi_{1m} \Rightarrow \phi_1')$ and $\phi_2 \leq \phi_1$

(3) $\phi = \phi_1, \phi_2$ otherwise

From the definition of $\text{pack}$ and $\text{pack}_2$, the computation of the overall type of the dot composite expression $A_1.A_2.1.true$ proceeds as follows:

$$\text{pack}(T_2 \Rightarrow T_1, \text{int} \Rightarrow T_2, \text{int}, \text{bool})$$
$$\Leftrightarrow \text{pack}_2(\text{pack}_3(\text{pack}_2(T_2 \Rightarrow T_1, \text{int} \Rightarrow T_2), \text{int}), \text{bool})$$
$$\Leftrightarrow \text{pack}_2(\text{pack}_3(\text{int} \Rightarrow T_1, \text{int}), \text{bool})$$
$$\Leftrightarrow \text{pack}_2(T_1, \text{bool})$$
$$\Leftrightarrow T_1, \text{bool}$$

where the number above $\Leftrightarrow$ indicates the $\text{pack}_2$ rule applied in the reduction.

### 6.2 Algorithm

The description of the type checking algorithm for the complete CSP language is beyond the scope of this paper. However, to demonstrate how type inference is conducted, process prefixing $c f_1 \cdot \cdots \cdot f_n \rightarrow P$ will be used. Process prefixing is special in that it spans both domains of the language. The action part $c f_1 \cdot \cdots \cdot f_n$ makes use of the expression language to construct messages in $f_1, \ldots, f_n$ communicated along the channel $c$ and the suffix $P$ is a process drawn from the process algebra. Also new variables may be introduced via input communication, however their scope does not extend beyond $P$.

Following the approach adopted in [6], rather than the direct application of the unification algorithm $\mathcal{U}(\phi_1, \phi_2)$ a procedure $\text{UNITY}(\phi_1, \phi_2)$ is used. This avoids the expensive multiple application of substitution to the type environment.

$\mathcal{U}$ and $\text{UNITY}$ are related as follows: $S$ and $S'$ represent the substitution before and after calling procedure $\text{UNITY}(\phi_1, \phi_2)$ and if $\mathcal{U}(S \phi_1, S \phi_2) = (\phi', S'')$ then $S' = S'' \circ S$.

The algorithm $\mathcal{W}$ produces a principle type $\phi$ for an expression $e$ in the presence of a type environment $\pi$ which holds the type bindings for each identifier visible in $e$. Here only the case when $e = c f_1 \cdot \cdots \cdot f_n \rightarrow P$ is considered.

**Algorithm $\mathcal{W}$**

$$\mathcal{W}(\pi, e) = \phi$$

let $\phi' = \mathcal{W}(\pi, c)$

let $\pi_0 = \pi$, and

for each $f_i (1 \leq i \leq n)$

if $f_i = ! e_i$: $\phi_i = \mathcal{W}(\pi_i \cdot e_i), \pi_i = \pi_i \cdot 1$

if $f_i = ? x_i$: $\phi_i = \alpha(\alpha$ is new), $\pi_i = \pi_i \cdot 1 \cup \{x_i \rightarrow \alpha\}$

$\mathcal{UNITY}(\phi', \text{Eq} \tau_1 \land \cdots \land \text{Eq} \tau_n)$

$$\text{pack}(\phi_1, \cdots, \phi_n) \Rightarrow \text{event}$$

$\mathcal{W}(\pi, P)$

$\text{UNITY}(\phi', \text{process})$

$\phi = \text{process}$

where

$$\pi_1 \cup \pi_2 = \pi_2 \cup (\pi_1 - \{[x/\phi] \mid [x/\phi] \in \pi_2 \cap [x/\phi] \in \pi_1\})$$

$\pi_1 \cup \pi_2$ in a channel $c$, the communication is said to be safe. The type $\tau_i$ (part of $\phi_i$) of each communicated value in $f_i$ is required to satisfy $\text{Eq} \tau_i$. Note the type environment used to type-check $P$ is the result of incrementing $\pi$ by new type bindings for variables introduced by input communications.

### 7 Related Work

The authors are aware of another CSP type checker created by Formal Systems (Europe) Limited. This work has been released in pre-release form for evaluation. Unfortunately no literature has been published and, as a result, no comparison to the work presented here can be made.
The introduction of qualified types is inspired by the work of Mark Jones on qualified types [3]. Compared with his general framework for a constrained type system, the approach described in this paper is tailored to suit the CSP language in order to simplify the decision procedure for determining type predicates.

In [7, 4] the type of processes (or behaviours [7]) contains information about the protocol types of all channels used in a process. As in CSP, channels may be hidden, renamed or even passed as parameters, it is not always possible for a static type-checker to keep track of the channels a process uses. As the purpose of type checking is to ensure that every communication is type safe it is not necessary to know what communication is actually performed in a process. Therefore, the type of processes is simplified by the basic type $\text{process}$.

8 Conclusion

In this paper a CSP type checking algorithm was presented that extends Milner’s polymorphic type checking algorithm capable of dealing with the particular combination of a process algebra and a functional language found in CSP. The algorithm forms the basis of ongoing work in data independence and is incorporated into the type checker of the Adelaide Refinement Checker (ARC) – a CSP based tool suite for model checking concurrent systems.

The unification algorithm presented in this paper does suffer one limitation. When it is presented with types such as $\text{int.} \alpha$ and $\alpha.\text{int}$ it will not terminate! This is because there are an infinite number of possible matches for $\alpha$ and the algorithm will attempt to find the most general form for $\alpha$. Fortunately, except in some pathological examples, this phenomenon rarely happens in practice. A solution to this limitation is actively being sought.

9 Acknowledgements

The authors would like to acknowledge the assistance of Charles Lakos, Jay Yantchev, Chen Gang and the anonymous reviewers.

References