A Theory of Core Fudgets

Colin J. Taylor
Languages and Programming Group
Department of Computer Science, University of Nottingham
University Park, Nottingham NG7 2RD, England.
http://www.cs.nott.ac.uk/~cjt

Abstract

The Fudgets system is a toolkit for developing graphical applications in the lazy functional programming language Haskell. In this paper we develop an operational semantics for a subset of this system, inspired by ideas from concurrency theory. A semantic theory based on bisimulation is defined and shown to be a congruence. We consider two applications of this theory: firstly, some equational rules useful for reasoning about Fudget programs are verified; secondly, we show how the operational semantics can be used to check the correctness of implementations of the Fudgets system.

1 Introduction

The Fudgets system [11] is one of a number of systems for building graphical applications in lazy functional programming languages such as Haskell [24]. A program in the Fudgets system is composed of a collection of *fudgets*, which correspond closely to concurrent processes in a process network. Fudgets can communicate with one another and also with the window system, and correspond to graphical components such as pushbuttons, windows and text-entry fields. Fudgets are based on the simpler concept of *stream processors* [15, 13], which cannot communicate with the window system. The Fudgets system includes combinators to connect stream processors together.

Numerous implementations of the Fudgets system exist, varying in the degree of concurrency they use. The original implementation [4] is written in Haskell and makes no use of concurrency. Alternatively, the Fudgets in Gadgets system [22] makes use of the concurrency provided by the Gadgets system [22].

In this paper we are concerned with formalising the meaning of programs written in the Fudgets system. This is useful for reasoning about such programs, and also for checking the correctness of implementations of the Fudgets system. Instead of attributing meanings to all the programs that can be developed using the Fudgets system, we only consider the subset that use stream processors. This is sufficient as fudgets can be encoded in terms of stream processors alone [5].

One approach to understanding the semantics of the Fudgets system is to use the semantic theory of the language in which the system is itself implemented. This is dependent on the particular implementation chosen, and may restrict the range of provable statements. For example, since the original implementation is deterministic we are unable to prove commutativity of the parallel composition of stream processors. Ideally, we require a semantics that abstracts away from such implementation details.

This paper develops a semantics for a stream processor sublanguage, called *Core Fudgets*, based on ideas from Milner's concurrent language CCS [16]. The semantics is a Plotkin style structured operational semantics [27], with the meaning of terms defined by a labelled transition system. Terms are equivalent if the transition trees induced by the semantics are bisimilar [23].

The main contributions of this paper are the definition of an operational semantics and associated semantic theory for the Core Fudgets language. The theory is used to verify some equational rules about stream processors that are useful for reasoning about programs. The operational semantics is also used to outline a method for checking the correctness of an implementation of Core Fudgets. These contributions form the basis for understanding the Fudgets system formally, and are described in more detail in the authors forthcoming thesis [29].

1.1 Related Work

We are aware of two other attempts to define a semantics for the Fudgets system, and we summarise the key points of these approaches. Our semantics resulted from an attempt to define the semantics of the Fudgets system in terms of the π-calculus [19], and we also review this approach.

A demand driven operational semantics for Fudgets: Hallgren and Carlsson [11] describe a simple operational semantics for the Fudgets system using rewriting rules. This semantics takes a demand driven approach, reducing stream processors that produce output before those that require input. Non-deterministic rewriting rules capture the concurrency of parallel composition of stream processors. The semantics we describe in this paper explores a different approach, inspired by concurrency theory, that has no bias towards generating output. The resulting semantics has more scope for concurrency and includes Hallgren and Carlsson's semantics as an instance.

A calculus for stream processors: More recently, Hallgren and Carlsson [5] have suggested a calculus for stream processors similar to the one we propose in this paper. However, their calculus is untyped and although they define an equivalence based on bisimulation they do not show it to be a congruence. They show how the λ-calculus can be encoded in their stream processor calculus, but this requires streams to be able to carry values of different types. This poses no problem for their untyped calculus but as a result we cannot use their encoding with our typed calculus.

A π-calculus semantics for Fudgets: An alternative approach to understanding the semantics of the Fudgets system is to encode it in a language that already has a formal semantics. The π-calculus is an appropriate choice as it supports concurrency and higher-order communication, which are both important features of the Fudgets system. This encoding led to the development of the semantics in this paper, which avoids some of the problems of using the π-calculus. For example, the π-calculus encodings are often large and non-intuitive in comparison to our operational semantics. The interested reader is referred to the author's forthcoming thesis [29] for more details.

1.2 This paper

This paper develops an operational semantics for Core Fudgets, along with a corresponding semantic theory based on bisimulation [23] — a standard notion of equivalence for concurrent processes. Our theory of Core Fudgets is similar to the theory of core CML [6], which is a concurrent version of the ML language. Bisimulation has also been used to define operational theories of functional languages [10], and for developing a theory of I/O in these languages [9]. Similarly, the theory we develop here makes use of bisimulation to formalise the I/O behaviour of stream processors, but, in contrast also embraces concurrency. This yields a semantics that is truly concurrent, but that has possible functional implementations. The PICT programming language [25] is specified in a similar manner.

Although the equational rules we formally develop for the Fudgets system are unsurprising, there is almost no previous work published on developing such rules. The only description of any rules for the Fudgets system appears in an early tutorial [3], and describes one distributive rule. This is unfortunate as the rules we develop can be useful for practical reasoning about Fudget programs such as optimisation.

One method of proving an implementation correct with respect to an operational semantics is to use testing preorders [21]. These have previously been used to show correctness of an implementation of the PICT language [28], and we discuss a similar approach for our semantics.

In the next section we describe the stream processor subset of the Fudgets system. Section 3 formalises the syntax and types of this subset in a language called *Core Fudgets*. The semantics for this language is defined in Section 4, while the corresponding semantic theory is described in Section 5. Section 6 details some applications of the semantics, and we conclude by describing ideas for future work in Section 7.

2 Stream Processors

At the heart of the Fudgets system is a stream processing sublanguage. A stream is a lazy list of values of possibly infinite length, while a stream processor is a process that consumes an input stream of values to produce a corresponding output stream of values. The Fudgets system builds on the concept of a stream processor to obtain the concept of a fudget, which is a stream processor that can communicate with the window system. It is possible to encode fudgets in terms of stream processors by tagging values on streams to indicate if a value came from or goes to another fudget or the window system.

Streams are considered to only ever carry values of a single type, and a type system is used to ensure this invariant is always true. A stream processor of type $SP\ in\ out$, can consume values of type in and produce values of type out. The behaviour of a stream processor is specified by the sequence in which it reads input from its input stream and produces output on its output stream. There are numerous mechanisms for accomplishing this in a functional language, such as continuation passing style I/O [14], or by using monads [31]. The original Fudgets system adopts a continuation passing style and defines three Haskell functions:

$$\begin{array}{lcl} putSP & :: & out \to SP\ in\ out \\ getSP & :: & (in \to SP\ in\ out) \to SP\ in\ out \\ nullSP & :: & SP\ in\ out. \end{array}$$

The first function is used to produce output, the second to consume input and the third to terminate a stream processor. The term $putSP\ v\ k$ is a stream processor that produces the value v on its output stream and then continues as the stream processor k. The stream processor $getSP\ f$ reads a value x on its input stream and then continues as the stream processor $f\ x$. Finally, the term $nullSP$ corresponds to a terminated stream processor that doesn't produce output or consume input.

2.1 Example

A simple example of a stream processor is the identity stream processor. This reads a value on its input stream, outputs it on its output stream, and continues as itself:

$$\begin{array}{lcl} idSP & :: & SP\ in\ out \\ idSP & = & getSP\ (\backslash x \to (putSP\ x\ idSP)). \end{array}$$

Another example is a stream processor that only passes on every other value it receives on its input stream to its output stream.

$$\begin{array}{lcl} skip & :: & SP\ in\ out \\ skip & = & getSP\ (\backslash x \to (getSP\ (\backslash y \to putSP\ x\ skip)). \end{array}$$

2.2 Combinators

Stream processors can be combined either in series, in parallel or in a loop, as illustrated in Figure 1. In these pictorial representations, input is consumed at the right, while output is produced on the left. Two streams are merged on the output side in parallel composition, while the loop combinator merges two streams on the input side. Hallgren and Carlsson's semantics [11] define this merge operation to be non-deterministic, which is how we interpret it too. This decision requires the combinators to be intrinsic in the semantics, rather than derived constructs from the $putSP$, $getSP$, and $nullSP$ functions as in the original Fudgets implementation. Note that the input to the parallel composition of two stream processors is distributed to both stream processors, unlike in a process algebra, such as the π-calculus, where only one of the two processes would receive the input value. We do not consider the looping combinator in the rest of this paper due to space restrictions, but the theory can be readily extended to encompass it [29].

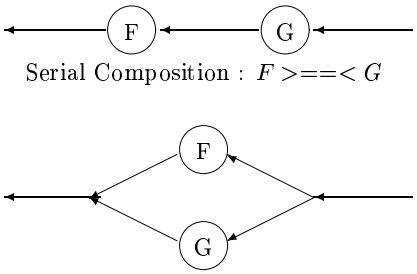


Figure 1: Stream Processor Combinators

3 Core Fudgets

The language we will consider in this paper, *Core Fudgets*, is a simplified version of the stream processor sublanguage of the Fudgets system. We do not include the λ-calculus in this language, which would allow arbitrary computation by stream processors. Such an extension is straightforward, and all the results stated in this paper hold for the extended language [29]. We define a type system for Core Fudgets, primarily since the original Fudgets system is implemented in a typed language. The type system also ensures that only values of one type can be communicated along a particular stream. The type system simplifies the semantics since we need not consider the meaning of terms that are ill-formed.

3.1 Types

We consider a monomorphic type system — polymorphism being an orthogonal issue for our semantics — with constant types and stream processor types. The symbol $\iota \in TConst$ ranges over a countable set of base types, with the set of all types described by the grammar:

$$\begin{array}{rcll} \tau & ::= & \iota & \textit{Constant Types} \\ & | & \mathbf{SP}\ \tau\ \tau' & \textit{Stream Processor Types.} \end{array}$$

3.2 Terms

We assume a countable set of term variables, $x, y, ... \in Var$, and a countable set, $c \in Const$, of constants, such that each constant c is assigned a type τ_c. The set of possibly open terms, denoted $E, F, ..., \in Expr^o$, is given by the grammar:

$$\begin{array}{rcll} E & ::= & x & \textit{Variables} \\ & | & c & \textit{Constants} \\ & | & \mathbf{NullSP}_{\tau\ \tau'} & \textit{Null Stream Processor} \\ & | & \mathbf{GetSP}\ x\ E & \textit{Input Stream Processor} \\ & | & \mathbf{PutSP}\ E\ F & \textit{Output Stream Processor} \\ & | & E >*< F & \textit{Parallel Composition} \\ & | & E >==< F & \textit{Serial Composition} \\ & | & E \ll F & \textit{Feed} \\ & | & \mathbf{FixSP}\ x\ E & \textit{Recursion.} \end{array}$$

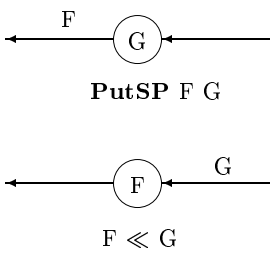


Figure 2: Duality of **PutSP** and $\ll$

There are some differences between the constructs listed here and those described in the previous section. Firstly, we introduce an explicit fixpoint construct, **FixSP**, used for defining recursive stream processors such as $idSP$. Secondly, **GetSP** and **FixSP** have an explicit variable argument as there is no representation for functions. These two constructs are similar to the corresponding constructs in the λ-calculus, but their typing rules are different. For example, the term **GetSP** $x\ x$ cannot be typed as this would allow a stream to carry values of differing types. Also, we only allow fixpoints to be defined over stream processor expressions.

A new construct called *feed* is introduced to specify the semantics of stream processors. It can be considered a dual to **PutSP** as shown in Figure 2. The intuitive meaning of $F \ll G$ is to add the term G to the input stream of the stream processor F. In the special case where we constrain our semantics to favour the production of output values, all instances of the feed construct can be eliminated. This construct corresponds closely to the $feedSP$ construct in the Chalmers implementation of the Fudgets system, which supports adding multiple terms to the input stream of a stream processor as opposed to just a single term.

We have a type indexed family of null stream processors, $\mathbf{NullSP}_{\tau\ \tau'}$,one for each combination of the types of input and output streams. For ease of reading, the type annotations on this construct will usually be omitted.

The constructs **GetSP** and **FixSP** are the only binding constructs, and free and bound variables are defined in the standard way. We denote the set of free variables in a term E as $FV(E)$, and the set of all closed terms as $Expr$. We define substitution in the standard way, and use $E[F/x]$ to denote substituting F for all free occurrences of x in E, where bound variables may be renamed in order to avoid capture of free variables. We only consider well-typed terms, defined inductively by the typing rules in Figure 3. A context, Γ, is a mapping from term variables to types. A typing judgement $\Gamma \vdash E : \tau$ asserts that the term E has type τ under the context Γ.

3.3 Examples

We conclude this section with some examples of stream processor programs in the Core Fudget language:

$$\begin{array}{rcl} idSP & = & \mathbf{FixSP}\ i\ (\mathbf{GetSP}\ x\ (\mathbf{PutSP}\ x\ i)) \\ skip & = & \mathbf{FixSP}\ s\ (\mathbf{GetSP}\ x\ (\mathbf{GetSP}\ y\ (\mathbf{PutSP}\ x\ s))). \end{array}$$

The first program is the identity stream processor, while the second is a program that removes every other value from its input stream.

$$(Var)\quad \frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau}$$

$$(Const)\quad \Gamma\vdash c:\tau_c$$

$$(Null)\quad \Gamma\vdash \mathbf{NullSP}_{\tau\ \tau'}:\mathbf{SP}\ \tau\ \tau'$$

$$(Input)\quad \frac{\Gamma, x:\tau\vdash E:\mathbf{SP}\ \tau\ \tau'}{\Gamma\vdash \mathbf{GetSP}\ x\ E:\mathbf{SP}\ \tau\ \tau'}$$

$$(Output)\quad \frac{\Gamma\vdash E:\tau'\quad \Gamma\vdash F:\mathbf{SP}\ \tau\ \tau'}{\Gamma\vdash \mathbf{PutSP}\ E\ F:\mathbf{SP}\ \tau\ \tau'}$$

$$(Series)\quad \frac{\Gamma\vdash E:\mathbf{SP}\ \tau'\ \tau''\quad \Gamma\vdash F:\mathbf{SP}\ \tau\ \tau'}{\Gamma\vdash E>==<F:\mathbf{SP}\ \tau\ \tau''}$$

$$(Parallel)\quad \frac{\Gamma\vdash E:\mathbf{SP}\ \tau\ \tau'\quad \Gamma\vdash F:\mathbf{SP}\ \tau\ \tau'}{\Gamma\vdash E>*<F:\mathbf{SP}\ \tau\ \tau'}$$

$$(Feed)\quad \frac{\Gamma\vdash E:\mathbf{SP}\ \tau\ \tau'\quad \Gamma\vdash F:\tau}{\Gamma\vdash E\ll F:\mathbf{SP}\ \tau\ \tau'}$$

$$(Fix)\quad \frac{\Gamma, x:\mathbf{SP}\ \tau\ \tau'\vdash E:\mathbf{SP}\ \tau\ \tau'}{\Gamma\vdash \mathbf{FixSP}\ x\ E:\mathbf{SP}\ \tau\ \tau'}$$

Figure 3: Typing Rules for Stream Processors

4 An Operational Semantics

In this section we define a structured operational semantics for Core Fudgets using ideas from Milner's concurrent language CCS [16]. We choose to give an operational semantics rather than a denotational semantics as we believe it is more intuitive and helps to give insight into the different implementations of the Fudgets system. In particular co-induction [20, 26] can be used to readily prove properties about recursive stream processor programs.

The semantics is defined in terms of a labelled transition system that characterises the immediate observations of Core Fudget programs. A labelled transition system is a set S of states, a set A of actions used to label transitions, and a transition relation, $\xrightarrow{\alpha}\subseteq S\times S$, for each $\alpha\in A$. We extend this definition to also include a typing context, and define the transition relation as $\Gamma\vdash E:\tau\xrightarrow{\alpha}F$ meaning that $\Gamma\vdash E:\tau$ and E can evolve into F by performing the action α. The typing context is required to ensure that only values of the correct type are communicated along streams.

4.1 Actions

The actions in our semantics represent the observations that we can make of Core Fudget terms. We consider three different kinds of action α as follows:

1. $\alpha = ?E$, an *input* action. A transition $\Gamma\vdash F:\tau\xrightarrow{?E}G$ means that F can evolve into G by receiving the term E on its input stream.

2. $\alpha = !E$, an *output* action. A transition $\Gamma\vdash F:\tau\xrightarrow{!E}G$ means that F can evolve into G by emitting the term E on its output stream.

3. $\alpha=\nu$, an *internal administrative* action. This action corresponds to the silent action τ of CCS. Here we use ν to avoid any confusion with the type τ. A transition $\Gamma\vdash F:\tau\xrightarrow{\nu}G$ means that F can evolve to G and requires no interaction with the external environment. Internal administrative actions arise from communications between stream processors.

Analogously to Core Fudget terms, we give typing rules for actions in Figure 4. The (*Red Action*) rule states that all internal administrative actions are always well-typed. The rule for input actions associates the type of the term input with the input action. Similarly, in the case of output actions, the type of the term being output is assigned to the output action. It is important to type actions as their type is used to constrain the transitions that terms can make.

$$(Red\ Action)\quad \Gamma\vdash\nu:\tau$$

$$(Input\ Action)\quad \frac{\Gamma\vdash E:\tau}{\Gamma\vdash ?E:\tau}$$

$$(Output\ Action)\quad \frac{\Gamma\vdash E:\tau}{\Gamma\vdash !E:\tau}$$

Figure 4: Type rules for Actions

4.2 Transitions

We define the transition relation explicitly, as opposed to defining a reduction relation and a structural congruence relation in the style of Milner [17, 18]. This simplifies proving properties of the transition relation as it is defined inductively by a set of inference rules for each possible form of Core Fudget term.

Variables, constants and **NullSP** have no transitions, as they cannot evolve further. Terms built from **GetSP** and **PutSP** have transitions indicating their ability to perform input and output actions:

$$(Inp_1)\quad \frac{\Gamma\vdash ?E:\tau}{\Gamma\vdash \mathbf{GetSP}\ x\ F:\mathbf{SP}\ \tau\ \tau'\xrightarrow{?E}F[E/x]}$$

$$(Out)\quad \Gamma\vdash \mathbf{PutSP}\ E\ F:\mathbf{SP}\ \tau\ \tau'\xrightarrow{!E}F$$

Any terms comprising actions that are not mentioned in the left hand side of the transition will be considered to be universally quantified. The (Inp_1) rule illustrates the need to extend the normal definition of transitions with typing contexts. The hypothesis of this rule ensures that only terms corresponding to the type of the input stream τ can be read by the stream processor **GetSP** x F. If we did not have this hypothesis then it would be possible for a transition to result in a term that is not well-typed.

Next, we consider the feed construct, which has two separate transition rules. The first rule allows a value to be fed into a stream processor that can perform an input transition of the appropriate term:

$$(Feed_1)\quad \frac{\Gamma\vdash E:\mathbf{SP}\ \tau\ \tau'\xrightarrow{?F}E'}{\Gamma\vdash E\ll F:\mathbf{SP}\ \tau\ \tau'\xrightarrow{\nu}E'}$$

The second transition rule for the feed construct captures the ability of output and administrative actions to occur in the term which is being fed values:

$$(Feed_2)\quad \frac{\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E' \quad \alpha \in \{\nu, !G\}}{\Gamma \vdash E \ll F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E' \ll F}$$

Serial composition has three separate transition rules, the first of which allows the rightmost stream processor to perform input and administrative transitions. The second rule supports the leftmost stream processor performing output and administrative transitions:

$$(Ser_1)\quad \frac{\Gamma \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} F' \quad \alpha \in \{\nu, \Gamma G\}}{\Gamma \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha} E >==< F'}$$

$$(Ser_2)\quad \frac{\Gamma \vdash E : \mathbf{SP}\ \tau'\ \tau'' \xrightarrow{\alpha} E' \quad \alpha \in \{\nu, !G\}}{\Gamma \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\alpha} E' >==< F}$$

The final transition rule for serial composition corresponds to the communication of data between the two stream processors. This can occur when the rightmost stream processor can perform an output transition, and the value output is then fed into the leftmost stream processor using the feed construct:

$$(Ser_3)\frac{\Gamma \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!G} F'}{\Gamma \vdash E >==< F : \mathbf{SP}\ \tau\ \tau'' \xrightarrow{\nu} (E \ll G) >==< F'}$$

Serial composition corresponds closely to parallel composition in a process calculi such as CCS. However, the (Ser_3) rule differs from the rule for synchronisation in CCS of a process that can perform an output with one that can perform an input. In particular, the leftmost stream processor in (Ser_3) does not need to be able to perform an input. The rule is modelling asynchronous output as the rightmost stream processor can perform its output action regardless of whether the leftmost stream processor is ready to consume input or not. The terms output from the rightmost stream processor are stored in the stream until the leftmost stream processor is ready to process them. The stream is thus modelled as a buffer. Here, we use the feed construct to build this buffer, as successive outputs from the rightmost stream processor will result in a chain of feed constructs into the leftmost stream processor. This allows the two stream processors to proceed at different rates.

We now turn our attention to the parallel composition of stream processors. Our first transition rule for parallel composition allows the leftmost stream processor to perform output and administrative transitions:

$$(Par_1)\quad \frac{\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E' \quad \alpha \in \{\nu, !G\}}{\Gamma \vdash E >*< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E' >*< F}$$

A symmetric version of this rule is also required, allowing the rightmost stream processor to perform output and administrative transitions:

$$(Par_2)\quad \frac{\Gamma \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} F' \quad \alpha \in \{\nu, !G\}}{\Gamma \vdash E >*< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E >*< F'}$$

All that remains is to consider when a parallel composition can perform an input transition. This case is quite subtle, and first we review the informal meaning of parallel composition. Values on the input stream to a parallel composition are received by *both* of the stream processors. This causes a problem if we attempt to define our transition rule similarly to the rule for parallel composition in CCS. The obvious transition rule is defined as:

$$\frac{\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'}{\Gamma \vdash E >*< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E' >*< F}$$

but this only allows one of the branches of the parallel composition to process the input value. The semantics we desire is for both of the branches to be able to process the input value. We can achieve this by buffering any input values on the branch not responsible for the input using the feed construct:

$$(Par_3)\quad \frac{\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E'}{\Gamma \vdash E >*< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} E' >*< (F \ll G)}$$

A symmetric version of this rule is also required, allowing the rightmost stream processor to perform an input action:

$$(Par_4)\quad \frac{\Gamma \vdash F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} F'}{\Gamma \vdash E >*< F : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?G} (E \ll G) >*< F'}$$

Finally we need to consider the semantics of recursion. Recursion is unwound by substituting the recursive term for the recursive variable as necessary. If a transition is possible when we unwind the recursion by one level then the overall recursive term can also make the same transition:

$$(Rec)\quad \frac{\Gamma \vdash E[(\mathbf{FixSP}\ x\ E)/x] : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E'}{\Gamma \vdash \mathbf{FixSP}\ x\ E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{\alpha} E'}$$

We conclude this section by proving some properties of the transition relation. In particular, we show that if a term can perform an output action then the type of the action must correspond to the type of the output stream of the term. A similar result holds for input actions, and we use both of these results to show a standard subject reduction property guaranteeing that the type of a term is preserved by the transition relation:

Lemma 1 (Subject Reduction)

1. *if* $\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!F} E'$ *then* $\Gamma \vdash F : \tau'$;
2. *if* $\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E'$ *then* $\Gamma \vdash F : \tau$;
3. *if* $\Gamma \vdash E : \tau \xrightarrow{\alpha} E'$ *then* $\Gamma \vdash E' : \tau$.

Proof. *Parts (1) and (2) result from a simple induction on the inference of the transitions* $\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{!F} E'$ *and* $\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?F} E'$, *respectively. The last part follows by induction on the inference of* $\Gamma \vdash E : \tau \xrightarrow{\alpha} E'$, *and by using parts (1) and (2) along with the standard substitution lemma which is easily proved* ∎

4.3 Examples

Before we consider formalising a theory for Core Fudgets, we pause for some examples, illustrating the operational semantics and its non-deterministic nature. The first example is given in Figure 5 and illustrates a simple stream processor producing two constants on its output stream. Figure 6 shows the second example, a stream processor that discards the second value it is sent, but forwards the first value sent to it. There is an infinite family of initial transitions, one for each possible instantiation of the input variable x. We only show one of the possible transitions for an arbitrary instantiation of x, and take this approach in the remaining examples too.

As an application of recursion, Figure 7, shows the meaning attributed to the identity stream processor. It is clear that the identity stream processor must output any value read before continuing to read more input.

The last example, in Figure 8, demonstrates the non-deterministic nature of the semantics for serial composition of stream processors. The initial term has two possible transitions as the term can either perform an output or read some input.

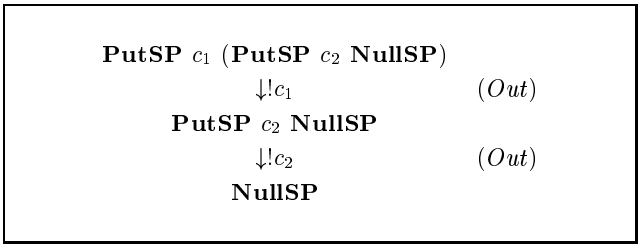


Figure 5: Output Example

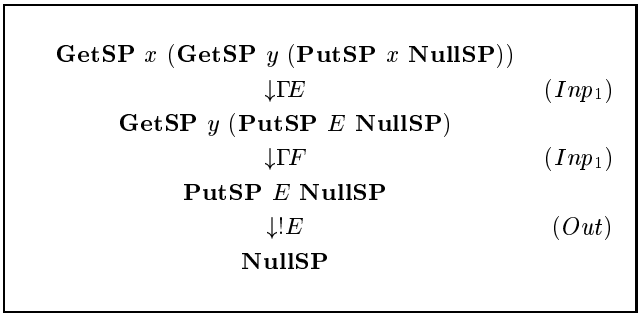


Figure 6: Discard Example

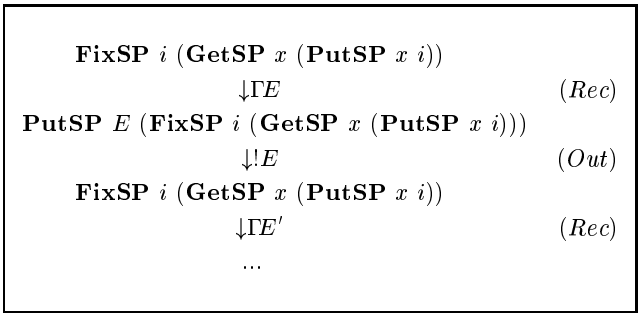


Figure 7: Identity Stream Processor Semantics

5 A Semantic Theory

We now focus our attention on defining an appropriate equivalence for Core Fudget terms. A higher-order variation of weak early bisimulation equivalence is the one that we finally choose. Intuitively, this compares terms of the language ignoring internal administrative actions, and comparing the terms sent along streams using the variation of weak early bisimulation equivalence itself.

We require our equivalences to only relate terms of the same type. This can be formalised by defining the equivalences as type-indexed families of relations:

Definition 1 (Type-indexed Relation) *A family of relations $\mathcal{R}_{?,\tau}$ between terms of the same type τ and context Γ, is a type-indexed relation if $E\,\mathcal{R}_{?,\tau}\,F$ implies that $\Gamma \vdash E : \tau$ and $\Gamma \vdash F : \tau$. We shall write $\Gamma \vdash E\,\mathcal{R}\,F : \tau$ to mean that $E\,\mathcal{R}_{?,\tau}\,F$, and will omit any obvious type information.*

Initially, we will consider equivalences on closed terms only, and so will formulate them as closed type-indexed relations. These equivalences are extended to open terms by considering all closing substitutions:

Definition 2 (Open Extension) *For a typing context Γ where $\Gamma = x_1 : \tau_1, ..., x_n : \tau_n$, then a Γ-Closure is a substitution $\cdot[\vec{G}/\vec{x}]$ where $\emptyset \vdash G_i : \tau_i$ for each i. The open extension, $\mathcal{R}^\circ$, of a closed type-indexed relation, $\mathcal{R}$, is the least type-indexed relation such that:*

- *$\Gamma \vdash E\,\mathcal{R}^\circ\,F : \tau$ if and only if for all Γ-closures $\cdot[\vec{G}/\vec{x}]$ then $\Gamma \vdash E[\vec{G}/\vec{x}]\,\mathcal{R}\,F[\vec{G}/\vec{x}] : \tau$.*

To form the basis of a useful semantic theory, an equivalence must be preserved by all of the constructs in the language. An equivalence relation with this property is a congruence, and we capture this formally by using contexts:

Definition 3 (Context) *A context is an expression containing a single hole, and we will use the metavariable $\mathcal{C}$ to range over them. We denote a context as $\Gamma \vdash \mathcal{C}[-_\tau] : \tau'$, where the hole satisfies the typing judgement $\Gamma \vdash -_\tau : \tau$. The instantiation of a context with a term E of the appropriate type, $\mathcal{C}[E]$, corresponds to filling the hole in $\mathcal{C}$ with the term E, which may capture variables free in E.*

Intuitively, an equivalence relation is a congruence if for any two terms that are related by the equivalence then the terms resulting from substituting these two terms into any context are also equivalent:

Definition 4 (Congruence) *A type-indexed equivalence relation, $\mathcal{R}$, is a congruence if it satisfies the following inference rule:*

$$(Cong\ \mathcal{R}) \quad \frac{\Gamma \vdash \mathcal{C}[-_\tau] : \tau' \quad \Gamma, \Gamma' \vdash E\mathcal{R}E' : \tau}{\Gamma \vdash C[E]\mathcal{R}C[E'] : \tau'}$$

where Γ' is the list of bound variables in the context $\mathcal{C}$.

This definition corresponds directly to the intuition required for an equivalence to be useful for equational reasoning. It states that if two terms are related by the equivalence then there is no context that can tell the two terms apart, and so we are free to replace one term by the other.

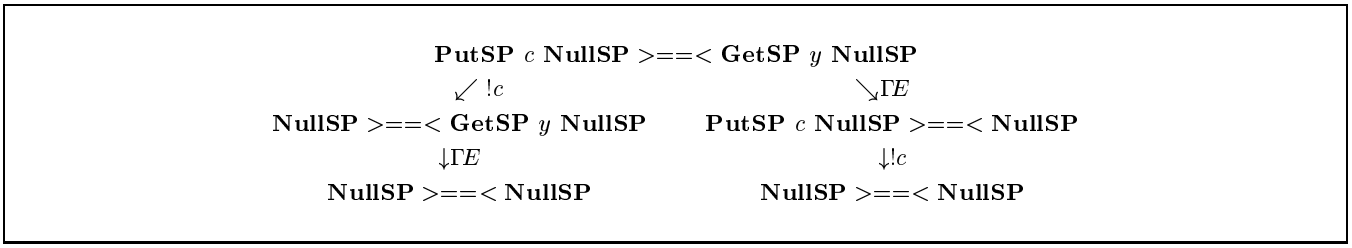


Figure 8: Mixed Input and Output Example

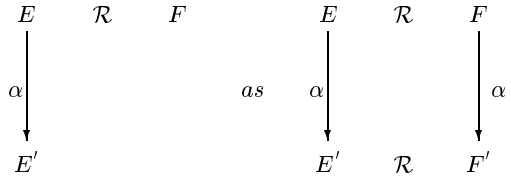
5.1 First-Order Bisimulation

Our starting point for an analysis of appropriate equivalences for Core Fudgets is *strong, first-order* bisimulation. We extend the usual definition of this equivalence to typed-indexed relations as follows:

Definition 5 (Strong First-Order Simulation) *$\mathcal{R}$ is a strong first-order simulation if it is a closed, type-indexed family of relations where $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *whenever $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F'.\emptyset \vdash F : \tau \xrightarrow{\alpha} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form, where types have been omitted for simplicity:



Definition 6 (Strong First-Order Bisimulation) *$\mathcal{R}$ is a strong first-order bisimulation if it is a strong first-order simulation and also symmetric. Strong first-order bisimilarity, $\sim^1$, is defined to be the greatest such relation.*

Proposition 1 *$\sim^1$ is an equivalence.*

Proof. *By definition $\sim^1$ is symmetric, and it is easy to show $\{(E,E) \mid E \sim^1 E\}$ to be a strong first-order bisimulation thus proving reflexivity. Similarly, we prove transitivity by showing that $\sim^1 \circ \sim^1$ is a strong first-order bisimulation.* ∎

We call this equivalence *first-order* as actions must be matched exactly. This requirement means that $\sim^1$ is not a congruence. It is easy to construct an example to demonstrate this as the equivalence is not preserved by the **PutSP** construct. For example, although we can show parallel composition to be commutative, $E >\!*\!< F \sim^1 F >\!*\!< E$, this equivalence is not preserved by the **PutSP** construct because **PutSP** $(E >\!*\!< F)$ $G \not\sim^1$ **PutSP** $(F >\!*\!< E)$ G.

This equivalence is *strong* as it requires internal administrative ν actions to be matched exactly. This yields a rather strict relation, and it is often the case that one wants to abstract away from the internal behaviour of terms. This corresponds to transitions of the form $E \xrightarrow{\nu} F$ being unobservable by the user. For example, considering the terms in Figure 9 we would intuitively expect A to be equivalent to B since both terms have an output transition. However, B must first make an internal administrative ν transition before it can make this output transition, and thus $A \not\sim^1 B$.

This problem can be solved by using the weaker form of bisimulation, referred to as *weak bisimulation*, which doesn't match ν transitions exactly. First, we define weak transitions which are used in the definition of weak bisimulation.

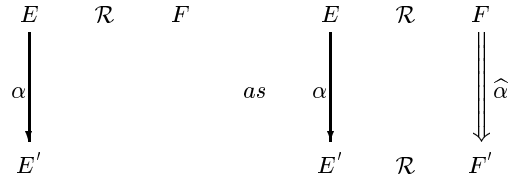
Definition 7 (Weak Transition) *Let $\Longrightarrow$ be the reflexive and transitive closure of $\xrightarrow{\nu}$, and $\stackrel{\alpha}{\Longrightarrow}$ be the composition $\Longrightarrow \circ \xrightarrow{\alpha} \circ \Longrightarrow$. We define a weak transition, $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ as:*

$$\stackrel{\widehat{\alpha}}{\Longrightarrow} = \begin{cases} \stackrel{\alpha}{\Longrightarrow} & \alpha \neq \nu \\ \Longrightarrow & \text{otherwise.} \end{cases}$$

Definition 8 (Weak First-Order Simulation) *$\mathcal{R}$ is a weak first-order simulation if it is a closed, type-indexed family of relations where $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *whenever $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F'.\emptyset \vdash F : \tau \stackrel{\widehat{\alpha}}{\Longrightarrow} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form, where types have been omitted for simplicity:



Definition 9 (Weak First-Order Bisimulation) *$\mathcal{R}$ is a weak first-order bisimulation if it is a weak first-order simulation and also symmetric. Weak first-order bisimilarity, $\approx^1$, is defined to be the greatest such relation.*

Proposition 2 *$\approx^1$ is an equivalence.*

Proof. *The proof follows similarly to Proposition 1.* ∎

5.2 Higher-Order Bisimulation

The first-order matching of actions results in equivalences that are too fine, as terms such as **PutSP** $(E >\!*\!< F)$ **NullSP** and **PutSP** $(F >\!*\!< E)$ **NullSP** are distinguishable. Here, we follow the solution taken by Thomsen [30], based on earlier work by Astesiano [1] and Boudol [2], where the terms output in an output transition must be bisimilar. This is called *higher-order bisimulation*, and we formalise it by first extending closed type-indexed relations to actions:

Definition 10 (Action Extension) *A type-indexed relation, $\mathcal{R}$, can be inductively extended to a relation over actions, $\mathcal{R}^a$, by the following rules:*

$$\frac{}{\nu\mathcal{R}^a\nu} \qquad \frac{}{\Gamma E\mathcal{R}^a\Gamma E} \qquad \frac{E\,\mathcal{R}\,F}{!E\mathcal{R}^a!F}$$

$$A = \mathbf{PutSP}\ c\ \mathbf{NullSP}$$
$$B = \mathbf{GetSP}\ x\ (\mathbf{PutSP}\ c\ \mathbf{NullSP}) >==< \mathbf{PutSP}\ c\ \mathbf{NullSP}$$

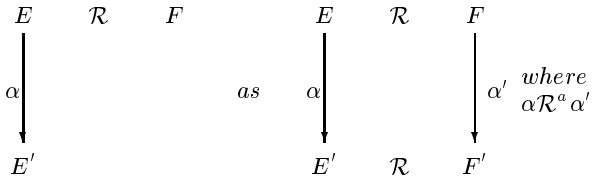
Figure 9: Example of terms not $\sim^1$ equivalent

Strong higher-order simulation is defined similarly to it's first-order counterpart:

Definition 11 (Strong Higher-Order Simulation) *$\mathcal{R}$ is a strong higher-order simulation if it is a closed, type-indexed family of relations where $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *if $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F', \alpha'.\emptyset \vdash F : \tau \xrightarrow{\alpha'} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$ and $\emptyset \vdash \alpha\ \mathcal{R}^a\ \alpha' : \tau'$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form, where types have been omitted for simplicity:



Definition 12 (Strong Higher-Order Bisimulation) *A strong higher-order simulation $\mathcal{R}$ is a strong higher-order bisimulation if it is symmetric. Strong higher-order bisimilarity, $\sim^h$, is defined to be the greatest such relation.*

Proposition 3 *$\sim^h$ is an equivalence.*

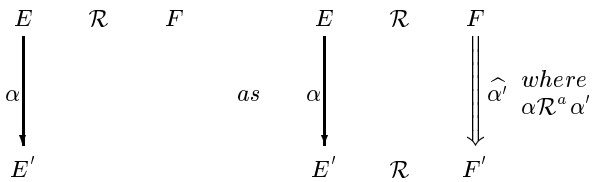
Proof. *The proof follows similarly to Proposition 1.* ∎

Strong higher-order bisimulation, $\sim^h$, is not particularly useful since the abundance of internal administrative ν transitions in the operational semantics stops us from showing even the simplest of properties. However, the natural extension of this equivalence to use weak transitions is a much more interesting equivalence that allows us to prove such properties. We will use this equivalence as the basis of an equational theory for Core Fudgets, and define weak higher-order simulation in an analogous manner to its first-order counterpart:

Definition 13 (Weak Higher-Order Simulation) *$\mathcal{R}$ is a weak higher-order simulation if it is a closed, type-indexed family of relations where $\emptyset \vdash E\mathcal{R}F : \tau$ implies:*

- *if $\emptyset \vdash E : \tau \xrightarrow{\alpha} E'$ then $\exists F', \alpha'.\emptyset \vdash F : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} F'$ and $\emptyset \vdash E'\mathcal{R}F' : \tau$ and $\emptyset \vdash \alpha\ \mathcal{R}^a\ \alpha' : \tau'$.*

Intuitively, we can think of this definition as requiring that we can complete all diagrams of the following form, where types have been omitted for simplicity:



Definition 14 (Weak Higher-Order Bisimulation) *$\mathcal{R}$ is a weak higher-order bisimulation if it is a weak higher-order simulation and also symmetric. Weak higher-order bisimilarity, $\approx^h$, is defined to be the greatest such relation.*

Proposition 4 *$\approx^h$ is an equivalence.*

Proof. *The proof follows similarly to Proposition 1.* ∎

This equivalence is our final candidate for the basis of an equational theory for Core Fudgets. As such it should be the case that the extension of this equivalence to open terms, $\approx^{h\circ}$, is a congruence:

Proposition 5 (Congruence) *$\approx^{h\circ}$ is a congruence.*

Proof. *The higher-order nature of the Core Fudgets language complicates this proof. The full proof [29] follows by using Howe's method [12], which works particularly well for higher-order languages.* ∎

6 Applications

In this section we illustrate some applications of the semantic theory developed for Core Fudgets. We use the theory to prove some useful equational rules and also outline a method for checking the correctness of an implementation of Core Fudgets.

6.1 Equational Rules

Instead of using the machinery of higher-order weak bisimilarity for reasoning about Core Fudget programs, it would be easier if we developed a set of simple equational rules:

Proposition 6 (Equational Rules) *The equational rules in Figure 10 hold for all $E, F, G \in Expr$.*

Proof. *The proof of these rules is routine. The two associative rules $(Assoc_1)$ and $(Assoc_2)$ make use of the distributive rules for the feed construct, $(Dist_1)$ and $(Dist_2)$.* ∎

These rules can be used to optimise Core Fudget programs. An obvious example is the (Id_1) rule which can be used to eliminate terminated stream processors corresponding to a kind of garbage collection. Similarly, the (Id_2) and (Id_3) rules can also be used to eliminate stream processors which have become 'stuck'.

Intuitively, it seems reasonable to expect that a serial composition of a stream processor, A, with the identity stream processor is just equivalent to A alone. However, this is not the case as we have that:

$$E >==< idSP \quad \not\approx^h \quad E$$
$$idSP >==< F \quad \not\approx^h \quad F$$

Considering the first of these rules then if we take E to be **NullSP** then the left hand side of the rule can perform an input transition that the right hand side cannot match.

$$
\begin{array}{lrcl}
(Zero) & \mathbf{NullSP} \ll E & \approx^h & \mathbf{NullSP} \\
(Dist_1) & (E >\!*\!< F) \ll G & \approx^h & (E \ll G) >\!*\!< (F \ll G) \\
(Dist_2) & (E >\!==\!< F) \ll G & \approx^h & E >\!==\!< (F \ll G) \\
(Dist_3) & (E >\!*\!< F) >\!==\!< G & \approx^h & (E >\!==\!< G) >\!*\!< (F >\!==\!< G) \\
(Comm) & E >\!*\!< F & \approx^h & F >\!*\!< E \\
(Assoc_1) & (E >\!*\!< F) >\!*\!< G & \approx^h & E >\!*\!< (F >\!*\!< G) \\
(Assoc_2) & (E >\!==\!< F) >\!==\!< G & \approx^h & E >\!==\!< (F >\!==\!< G) \\
(Id_1) & \mathbf{NullSP} >\!*\!< E & \approx^h & E \\
(Id_2) & \mathbf{NullSP} >\!==\!< \mathbf{PutSP}\ E\ F & \approx^h & \mathbf{NullSP} \\
(Id_3) & \mathbf{GetSP}\ x\ E >\!==\!< \mathbf{NullSP} & \approx^h & \mathbf{NullSP}
\end{array}
$$

Figure 10: Core Fudget Equational Rules

The problem is evident for a term that has both an immediate input and output transition, as we can observe both actions. If instead, we constrain our observations of serial and parallel compositions, such that we can only observe input actions when there are no output actions to observe, then the rules become valid. Interestingly, this constraint also eliminates the need for the feed construct, since an input can only occur for a parallel composition when both branches are ready to perform an input. In the case of serial composition any values communicated between the two stream processors must be processed immediately in case they result in the ability to produce some output, and thus again the feed construct is not required. This approach is the one taken in the original Fudgets implementation.

One possibility for solving this problem such that the above identity rules do hold is to add the following inference rule to the operational semantics:

$$(Inp_2) \quad \frac{\Gamma \vdash F : \tau}{\Gamma \vdash E : \mathbf{SP}\ \tau\ \tau' \xrightarrow{?\mathrm{F}} E \ll F}$$

For the first identity rule, if $E = \mathbf{NullSP}$ then the right hand side can now make a matching input transition as required.

The feed construct must be an intrinsic part of the language, rather than being defined in terms of serial composition and the identity stream processor, because these identity rules do not hold. If we defined feed as:

$$E \ll F \stackrel{\mathrm{def}}{=} E >\!==\!< \mathbf{PutSP}\ F\ idSP.$$

then this would lead to unexpected behaviour. For example, the stream processor in Figure 11 initially consumes an item of data but then the resulting term can make another input transition due to the definition of feed in terms of $idSP$.

$$
\begin{array}{c}
(\mathbf{GetSP}\ x\ \mathbf{NullSP}) >\!*\!< \mathbf{NullSP} \\
\downarrow \Gamma E \\
\mathbf{NullSP} >\!*\!< (\mathbf{NullSP} \ll E) \\
\downarrow \Gamma E' \\
\ldots
\end{array}
$$

Figure 11: Problem with Feed

6.2 Correctness of Implementations

Our operational semantics is also useful for checking the correctness of Fudget implementations. In order to check the correctness of an implementation we need to consider the implementation as a labelled transition system itself. If we can show this labelled transition system to *simulate* the labelled transition system of the operational semantics then we consider the implementation to be correct. Simulation in this case captures two properties:

- *Linear Time*, the operational semantics is nondeterministic but most implementations will be deterministic. We do not intend our definition of correctness to prescribe particular scheduling strategies. Most practical implementations will be incorrect if we base correctness on branching time and so we only require correctness to respect linear time.
- *Termination*, although we do not consider an implementation which does not exhibit all the concurrent computations captured by the operational semantics to be incorrect, we do insist on *maximal* executions. If a program terminates under the implementation then it must do so under the operational semantics too.

These properties lead us to propose a form of termination-preserving simulation for the definition of correctness. We begin by extending weak higher-order simulation to use two different transition relations, $\xrightarrow{\alpha}_i$, and the transition relation $\xrightarrow{\alpha}$ of the operational semantics in Section 4.

Definition 15 (Correctness Simulation) *A closed, type-indexed family of relations $\mathcal{R}$ is a correctness simulation if $\emptyset \vdash E \mathcal{R} F : \tau$ implies:*

- *if $\emptyset \vdash E : \tau \xrightarrow{\alpha}_i E'$ then $\exists F', \alpha'. \emptyset \vdash F : \tau \stackrel{\widehat{\alpha'}}{\Longrightarrow} F'$ and $\emptyset \vdash E' \mathcal{R} F' : \tau$ and $\emptyset \vdash \alpha\ \mathcal{R}^a\ \alpha' : \tau'$.*

Definition 16 (Termination-Preserving) *A closed type-indexed family of relations, $\mathcal{R}$, is termination-preserving if $\emptyset \vdash E \mathcal{R} F : \tau$ and $\emptyset \vdash E : \tau \not\rightarrow_i$ implies $\exists F'. \emptyset \vdash F : \tau \Longrightarrow F'$ where $\emptyset \vdash F' : \tau \not\rightarrow$.*

Definition 17 (Correctness) *An implementation of Core Fudgets characterised by a transition relation $\xrightarrow{\alpha}_i$ is correct with respect to the operational semantics if there is some termination-preserving correctness simulation, $\mathcal{R}$, such that for all $E \in Expr$ then $\emptyset \vdash E \mathcal{R} E : \tau$.*

7 Conclusions

We have considered the stream processing sublanguage of the Fudgets system, and described a corresponding operational semantics based on concepts from concurrency theory. A semantic theory has been developed using the concept of bisimulation, and was used to prove some simple equational rules. Finally, we concluded by showing how the operational semantics can be used to check the correctness of implementations of this language.

The semantics presented here has been motivated from a concurrency perspective. As a result we considered serial composition of stream processors in a parallel manner unlike Hallgren and Carlsson's semantics [11]. In order to specify the correct semantics it is necessary to introduce a special construct, feed. When we constrain this semantics to be biased towards generating output then we obtain the semantics given by the existing Haskell-like rewriting rules, and can eliminate the feed construct entirely.

As previously mentioned, another approach for specifying the semantics of the Core Fudgets language is to encode it in a language such as the π-calculus. This may be particularly interesting as it could be used to compare the Fudgets system with other graphical systems, like Haggis [7], which already have π-calculus semantics.

It would be interesting to try and completely axiomatise the semantic theory of Core Fudgets. Such an axiomatisation would be particularly useful from a practical perspective, as the tedious machinery of bisimulation could be completely avoided.

There are other implementations of the Fudgets system, and verification of them with respect to our semantics may lead to interesting insights into the relationships between them. Parallel implementations of the Fudgets system could also be explored. One possibility would be to use Concurrent Haskell [8] as the basis of a concurrent implementation.

One of the problems with the theory presented here is that when arbitrary computation is allowed in stream processors then the degree to which we can reason about them may be severely hampered. The stream processor:

$$\textbf{GetSP}\ x\ (\textbf{if}\ x < 3\ \textbf{then PutSP}\ \mathit{True}\ \textbf{NullSP}\ \textbf{else NullSP}),$$

illustrates this, as we can determine that it will consume an input value, but not whether it will produce an output value. This can only be determined if we know the input to the stream processor.

Acknowledgements

This work was supported by a University of Nottingham studentship. I would also like to thank my colleagues in the Languages and Programming research group at the University of Nottingham, particularly Graham Hutton, for their valuable contributions they have made to the work described in this paper. The addition of the feed construct was first suggested to me by Magnus Carlsson and Thomas Hallgren, while the addition of the (Inp_2) rule to the operational semantics was suggested by Andrew D. Gordon. Finally, thanks go to the referees who provided detailed and helpful comments.

References

[1] E. Astesiano, A. Giovini, and G. Reggio. Generalized bisimulation in relational specification. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, volume 294 of *Lecture Notes in Computer Science*, pages 207–226. Springer-Verlag, 1988.

[2] G. Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT '89*, volume 351 of *LNCS*, pages 149–161, 1989.

[3] M. Carlsson and T. Hallgren. Programming with fudgets. Available by anonymous FTP from `pub/users/hallgren` on `ftp.cs.chalmers.se`., December 1994.

[4] M. Carlsson and T. Hallgren. The Fudget Library distribution. Available by anonymous FTP from `pub/haskell/chalmers` on `ftp.cs.chalmers.se`., 1995.

[5] M. Carlsson and T. Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 1998.

[6] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core cml. Computer Science Technical Report 95:05, School of Cognitive and Computing Sciences, University of Sussex, 1995.

[7] S. Finne and S. L. P. Jones. Composing Haggis. In *Eurographics Workshop on Programming Paradigms in Computer Graphics*, April 1995.

[8] S. Finne and S. L. P. Jones. Concurrent Haskell. In *Proceedings of the Twenty Third ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[9] A. D. Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, August 1992.

[10] A. D. Gordon. Bisimilarity as a theory of functional programming. Mini-course. BRICS Notes Series NS–95–3, Brics, Aarhus University, 1995. Extended version of MFPS'95 and Glasgow FP'94 papers.

[11] T. Hallgren and M. Carlsson. Programming with Fudgets. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer Verlag, May 1995.

[12] D. J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[13] G. Kahn. A preliminary theory for parallel programs. Rapport de Recherche 6, IRIA, January 1973.

[14] K. Karlsson. Nebula: A functional operating system. Technical report, Programming Methodology Group, Chalmers University of Technology and University of Gothenburg, 1981.

[15] P. Landin. A correspondence between algol 60 and church's lambda-notation: Part i and ii. In *Communications of the ACM*, volume 8, pages 89–101,158–165, February and March 1965.

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, 1985.

[17] R. Milner. Functions as processes. Research Report No. 1154, INRIA, February 1990.

[18] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report ECS-LFCS-89-85 and 86, Department of Computer Science, The University of Edinburgh, 1989.

[20] R. Milner and M. Tofte. Co-induction in relational semantics. In *Theoretical Computer Science*, volume 87, pages 209–220, September 1990.

[21] R. D. Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:73–133, 1984.

[22] R. Noble. *Lazy Functional Components for Graphical User Interfaces*. PhD thesis, Dept. of Computer Science, University of York, November 1995.

[23] D. M. Park. Concurrency on automata and infinite sequences. In *Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[24] J. Peterson and K. Hammond. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, May 1996.

[25] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[26] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[27] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, September 1981.

[28] P. Sewell. On implementations and semantics of a concurrent programming language. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405. Springer-Verlag, 1997.

[29] C. J. Taylor. *Formalising and Reasoning about Fudgets*. Ph.d. thesis, Department of Computer Science, University of Nottingham, To appear.

[30] B. Thomsen. *Calculi for Higher Order Communicating Systems*. Ph.d. thesis, Department of Computing, Imperial College, 1990.

[31] P. Wadler. The essence of functional programing (invited talk). In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–14, January 1992.